# 第一章 绪　言

## 1.0 引言

本书旨在向读者教授实用、有效且完美的数值计算方法。假定读者要完成一些特定的任务，我们将教你学会如何来进行此项任务。偶尔，我们会带你进入美丽的小径，但更多的时候，我们将引导你沿着通往目的地的大道前进。

通览全书，读者将会发现我们会无所顾忌地告诉你，应该怎样做和不应该怎样做。这种规定性的语调来源于我们清醒的决策，希望你不要为此而恼火。我们并不是说我们的建议绝对正确！相反，我们反对一种倾向：即在关于数值计算的教科书文献中，只讨论已经发明的各种可能方法，而不提及基于相对价值的实际评价。因此，我们尽可能将我们的实际评断提供给读者。当读者有了一定经验后，将会对我们建议的可信度形成自己的看法。

我们假定读者能够阅读 C 语言计算机程序。在这本《C 语言数值算法程序大全(Second Edition)》中，所有算法都用 C 语言实现。如果读者更乐意用 FORTRAN 语言，你将找到本书的 FORTRAN 版本——《Numerical Recipes in FORTRAN(Second Edition)》。本书其它语言的早期版本，例如 BASIC 和 Pascal，同样是完全有效的，只是不包含新版增加的内容。

在印刷形式上，列入文中的程序都用如下格式：

```
#include <math.h>
#define RAD (3.14159265/180.0)

Void flmoon (int n,int nph, long *jd, float *frac)
```

程序总以一个导引说明开始，总括程序的目的并解释调用的参数序列。本程序用来计算月相，给定一个整数 n 和所期望的月相的代码 nph(nph=0 为新月，nph=1 为上弦月，nph=2 为满月，nph=3 为下弦月)。程序返回从 1900 年 1 月起第 n 个月相发生的凯撒历日数 jd，并加上格林威治标准时的时分数 frac(它用一天的小数部分表示)。

```
{
    void nrerror(char error_text[]);
    int i;
    float am,as,c,t,t2,xtra;

    c=n+nph/4.0;                              这是我们对单行注解的方法
    t=c/1236.85;
    t2=t*t;
    as=359.2242+29.105356*c;                  不必理解这个算法，但它却很有效!
    am=306.0253+385.816918*c+0.010730*t2;
    *jd=2415020+28L*n+7L*nph;
    xtra=0.75933+1.53058868*c+((1.178e-4)-(1.55e-7)*t)*t2;
    if (nph == 0 || nph == 2)
        xtra += (0.1734-3.93e-4*t)*sin(RAD*as)-0.4068*sin(RAD*am);
    else if (nph == 1 || nph == 3)
        xtra += (0.1721-4.0e-4*t)*sin(RAD*as)-0.6280*sin(RAD*am);
    else nrerror("nph is unknown in flmoon");      此处指示错误条件的方法
```

```
    i=(int)(xtra >= 0.0 ? floor(xtra) : ceil(xtra-1.0));
    *jd += i;
    *frac=xtra-i;
}
```

如果上面函数定义的语法对读者来说比较陌生的话，表明你可能习惯于使用 K & R 语法，而不是较新的 ANSI C 语法。在本书中，我们采用 ANSI C 语法作为我们的标准。读者可以翻到后面第 1.2 节，在那里我们对 ANSI C 函数的范例进行了更加详细的讨论。

注意，我们表示错误和例外情况的惯例是用一条象 nrerror("some error message")的语句。函数 **nrerror**()是列于书末附录 B 的实用程序 **nrutil.c** 中。附录 B 还包含了其它实用程序，我们将在后面介绍。函数 **nrerror**()在用户的 **stderr** 设备(通常是用户的屏幕终端)上印出错误信息，然后调用函数 **exit**()终止运行。在我们所检查过的任何 C 库中都有函数 **exit**()，但假若库中没有，可以修改 **nrerror**()，以使程序除暂停运行外，还可做其它一些事情。例如，可使程序暂停而从键盘输入数据，然后人为地中断运行。在一些应用中，读者可能想修改 **nrerror**()，以做一些更复杂的错误处理，如用错误标志或代码设置向其它地方传输控制。

我们将在第 1.1 节和 1.2 节中，更多地讨论 C 语言的协议和风格。

### 1.0.1 计算环境和程序有效性

我们的目的是，尽可能使本书中的程序通用性强一些，以便能在不同型号的计算机、不同的操作系统、不同的 C 编译系统上运用。C 实际上就是为了可移植性而设计的。但是，我们发现，只有在多种不同的编译器上，实际检查了所有的程序后，才能了解库结构和内容方面的差异。甚至会发现，所容许的语法有区别。我们已在表 1.0.1.1 所列的机型、操作系统和编译器的各类组合下，检验了本书的所有程序。更一般地说，各程序可以不加修改地在任何符合标准 ANSI C 的编译器上运行。标准 ANSI C 在哈宾逊(Harbison)和斯蒂尔(Steele)[1]所著的书中叙述得相当出色。经过较少的改动后，我们的程序也可在较老的 K & R 标准[2]的编译器上运行。一个不起眼的、但必须注意的不兼容的例子是，ANSI C 中，内存分配函数 malloc()和 free()要求通过标题文件 stdlib.h 说明；而一些较老的编译器要求它们通过标题文件 malloc.h 说明；还有一些编译器将它们看做内部文件而不要求任何标题文件说明。

**表 1.0.1.1 经过检验的机器和编译器**

| 硬 件 | 操作系统版本 | 编译器版本 |
|---|---|---|
| IBM PC compatible 486/33 | MS-DOS 5.0/Windows 3.1 | Microsoft C/C++7.0 |
| IBM PC compatible 486/33 | MS-DOS 5.0 | Borland C/C++2.0 |
| IBM RS/6000 | AIX 3.2 | IBM xlc 1.02 |
| DECstation 5000/25 | ULTRIX 4.2A | CodeCenter (Saber) C 3.1.1 |
| DECsystem 5400 | ULTRIX 4.1 | GNU C Compiler 2.1 |
| Sun SPARCstation 2 | SunOS 4.1 | GNU C Compiler 1.40 |
| DECstation 5000/200 | ULTRIX 4.2 | DEC RISC C2.1* |
| Sun SPARCstation 2 | SunOS 4.1 | Sun cc 1.1* |
| * 编译器版本不能完全实现 ANSI C，只对 K & R 有效。 | | |

为保证程序有效性，我们直接将机器可读的程序作为书稿中的程序源码，以减少印刷错

误繁殖的机会。有效的演示程序可从《C语言数值算法实例》一书得到，其中的程序亦是机器可读的。如果读者计划使用本书中较多的程序，或计划在一种以上不同的计算机上使用书中的程序，你将发现，有这些演示程序的拷贝是非常有用的。

当然，说我们的程序毫无错误将是非常愚蠢的，我们确实并没有这样说。虽然许多读者将他们的经验写信告诉了我们，使我们从中得益，但是我们还是非常谨慎的。读者若发现了错误，请证实它并告诉我们。

### 1.0.2　和本书第一版的兼容性

如果读者已习惯于本书第一版的数值算法程序，那么请记住：第一版的几乎所有程序都列入本书第二版内，并具有相同的名字和功能，通常是对程序代码本身做了一些改进。还有，我们希望读者能很快熟悉新版本中增加的100多个程序。

我们已经放弃了第一版中一小部分程序，而用我们认为本版中更好的程序来替代。在表(表1.0.2.1)中，我们列出了放弃的程序以及它们的替代程序。

表1.0.2.1　本版中省略的第一版中的程序

| 名　字 | 替换物 | 注　释 |
|---|---|---|
| adi | mglin or mgfas | 更好的方法 |
| cosft | cosft1 or cosft2 | 边界条件的选择 |
| cel, el2 | rf, rd, rj, rc | 更好的算法 |
| des, desks | ran4 现在用 psdes | 过去速度太慢 |
| mdian1, mdian2 | select, selip | 更一般 |
| qcksrt | sort | 名字改变 |
| rkqc | rkqs | 更好的方法 |
| smooft | 用convlv，其系数来自savgol | |
| sparse | linbcg | 更一般 |

本书第一版的使用者还应该认识到，两版都有的一些程序，其调用接口也有变化，因此并不是直接的“插入式兼容”。下面较全面地列出了此类程序：**chsone**、**chstwo**、**covsrt**、**dfpmin**、**laguer**、**lfit**、**memcof**、**mrqcof**、**mrqmin**、**pzextr**、**ran4**、**realft**、**rzextr**、**shoot**、**shootf**。还有一些其它共同的程序(这和采用本书第一版的哪一次印刷有关)。如果读者在第一版程序的基础上已经编写了一些较为复杂的软件，我们并不盲目地建议你用本书对应的程序代替它们。但是我们特别建议新的程序编写者采用本书的新程序。

### 1.0.3　参考书

在本书的大多数章节后列出了参考文献，以便读者进一步研究。在正文中引用的参考文献用括号中的数字，如[3]表示。

因为“计算机的算法”通常在正式发表之前已经流传了一段时间，因此要列出“原始文献”的任务比较艰巨。我们试图不这样做，我们也不想为了文献目录的完整性而做任何伪装。但有些主题存在相当多的第二类文献(教材、评论中的讨论)，那么，我们有意识地将这类主

题的参考文献限于少数的、非常有用的第二类文献，特别是参考价值较大的文献。当有些主题没有适当的第二类参考文献存在时，我们给出少数几个原始来源，它们可作为进一步阅读的起点，而不作为该领域完整的文献目录。

参考文献所列的顺序并不表示任何必然的意义。它只反映了按引用顺序列出参考文献和大致按优先级的先后列出进一步研究的参考建议，具体来说，就是最有用的列在最前面。

在本章的下面三节中，我们将复习程序设计（控制结构等）的基本概念，讨论一套我们在本书中采用的针对C的协议，并介绍一些数值分析的基本概念（舍入误差等）。然后，进入本书的实质内容。

参考文献和进一步读物：


Harbison, S. P., and Steele, G. L., Jr. 1991, C:*A Reference Manual*, 3rd ed. (Englewood Cliffs, NJ: Prentice-Hall). [1]

Kernighan, B., and Ritchie, D. 1978, *The C Programming Language* (Englewood Cliffs, NJ: Prentice Hall). [2] [Reference for K&R"traditional" C. Later editions of this book conform to the ANSI C standard.]

Meeus, J. 1982, *Astronomical Formulae for Calculators*, 2nd ed., revised and enlarged (Rich-mond, VA: Willmann-Bell). [3]


## 1.1 程序组织和控制结构

我们有时喜欢指出，计算机程序与赋诗和作曲之间的极其相似之处。所有这三者都是在二维纸面或计算机屏幕上以视觉媒质——符号出现。但在这三者中，可视的二维静态形式传送（或假设传送）着一些极不相同的东西，即随时间展开的过程不同。诗供阅读，音乐为演奏，程序作为计算机指令序列顺序地执行。

在这三者中，就传播的视觉形式而言，其目标是人。目的是最高效率地传送，进而使人最大程度地理解过程将如何随时间展开。就诗而言，其传送目标是读者；对音乐来说，其目标是演奏者；而程序的目标是程序用户。

读者可能会反驳说，程序的传送目标，不是人而是计算机，程序用户只是作为不相干的中介物——一个给机器供料的侍者。这也许反映了下面情形：市政当局的行政官员将软磁盘塞入他的个人计算机中，然后喂它一个二进制可执行的黑箱程序。在这种情况下，计算机并不管这个程序是否是"优秀编程策略"的程序。

然而，我们却要假设你——本书的读者是处于完全不同的情形下。你需要或想要知道，一个程序做的是什么，而且要知道它是如何做的。这样你就能够根据自己的特定用途调整和修改它。你需要其他人能了解你所做的一切，进而提出赞赏和批评。在这些情况下，一个程序传送的目标完全是人而非机器。

要达到程序具有优秀编程策略，其关键是认识音乐、诗和程序三者作为人脑的符号结构，都是自然地组成层次的，并且有许多不同的嵌套层。语音（音素）形成小的有意义的单元（词素），它又组成单词；单词组合为短语，进而连成句子；句子形成段落，再构造出更高而有意义的层次。音符组成音乐短语，进而组合成主旋律、配合旋律和和声等；再形成乐章、协奏曲、交响乐等等。

程序的结构同样是有层次的[1-3]。恰当地说，优秀编程策略能使不同技术产生不同的编程水平。程序中最低层是 ASCII 码字符集；然后是常量、标识符、操作数、运算符；接着是象 a[j+1]=b+c/3.0这样的程序语句。这里，对编程员最好的忠告是程序要简单明了，或者说不要过于花哨。如果想将 j=(0,1,2)时，其对应的 k=(1,2,0)的值顺序排列出来，你可能会因写出如下的单行语句而沾沾自喜：

```
k=(2-j)*(1+3*j)/2;
```

但是，当以后重新想读懂它时就会后悔了。更好的并且也可能是更快的程序应为：

```
k=j+1;
if  (k==3)  k=0;
```

某些编程专家们甚至赞成如下的程序：

```
switch (j){
    case 0: k=1; break;
    case 1: k=2; break;
    case 2: k=0; break;
    default: {
        fprintf (stderr, "unexpected value for j");
        exit (1);
    }
}
```

因为此程序很明了，而且能够有效地预防非期望的 j 值所引起的错误。在这些程序中，我们一般倾向于中间的那个程序。

在这个简单的示例中，我们实际上经历了几个层次。语句频繁地以“组”或“块”的形式出现，仅当它们作为一个整体时才能表示一定的意义。例如：

```
swap=a[j];
a[j]=b[j];
b[j]=swap;
```

任何程序员都会立刻领悟到，它是在交换两个变量，而

```
ans=sum=0.0;
n=1;
```

很象是在某些迭代过程前，对变量进行初始赋值。程序中的这一层通常是很明显的，优秀的编程策略是，在这一层放入注释，即加“初始赋值”或“交换变量”的说明。

下一层就是“**控制结构**”，这主要包括如上例中的 **switch** 结构、**for** 循环等等。这一层非常重要，并且和本书所列程序的层次有关，我们将讨论于下。

再更高的一层就是函数和模块，以及要完成计算任务的“全局”结构。和音乐类比，我们现在是处在乐章和完整的作品这一层。在这一层，“**模块化**”和“**封装**”成为重要的编程概念，一般可做如下理解：各程序单元应该通过明确的定义和严密限定的接口相互连在一起。好的模块化策略是大型复杂软件工程成功的先决条件，特别对多个编程者共同协作完成的程序更是如此。对于某位科学家或者本书的读者，即使所遇到的不是那么庞大的编程，这种模块化策略也是有益的(尽管不是特别必要)。

一些计算机语言，象Modula-2和C++，能够实现C中缺乏的高级语言结构模块。例如，在Modula 2中，函数、类型定义和数据结构都能"封装"成"模块"，通过公有接口进行通信，而它们的内部工作和程序的其余部份分离开并隐藏起来[4]。在C++语言中，重要的概念是"类(class)"，也就是用户定义的数据类型的推广，它提供数据的隐藏、数据自动初始化、内存管理、动态打印和运算符的重载(例如，用户对运算符，如+和×进行定义延伸，以便能进行某种特定的运算)[5]。如果"类"能正确地用来定义在不同的程序单元之间传递的数据结构，则它可使程序单元的公用接口更明晰、更确定，减少编程出错的机会，增加编译时和运行时的错误查找。

除了模块化，还有"**面向对象的程序设计**"概念，尽管此概念依赖于模块化。这时，一些编程语言，象C++或Turbo Pascal 5.5[6]，允许模块的公有接口接受类型或活动的重定义，这些重定义在本模块这一级都能被享用(因此被称为**多态性**)。例如，一个用来对实数矩阵求逆的程序能够在运行时，自动地用来对复数矩阵求逆，采用的是，复数数据类型重载和相应的算术运算定义的方法。还有两个概念也会碰到，即"**继承性**"(除了自己本身的结构功能外，还能"继承"其它类型结构而重定义数据类型的能力)和"**对象扩展**"(在运行时，不通过源码就直接对模块增加功能的能力)。

在本书中，我们没有对程序模块化，也没有对象化代码，这有两个原因：第一，所选用的C语言不能真正做到这一点。第二，我们预想读者能花一些时间，用自己选择的结构，将书中的算法模块化或对象化。目前，还没有标准的或可接受的面向科学对象计算的一些"类"。然而，当我们必然地将本书的算法内容和某些选取的与类定义有关的特定集合联系起来时，我们可能正在创建这种面向科学对象计算的一些"类"。

另一方面，我们不是不赞成模块化和面向对象的程序设计，但由于C的限制，我们只能做得尽可能使我们的程序面向对象化。这也是我们采用ANSI C的函数原型作为缺省的C语言的原因(第1.2节)。还有，我们在程序的各个实施部分，特别注意**结构化的程序设计**。如下面我们所讨论的。

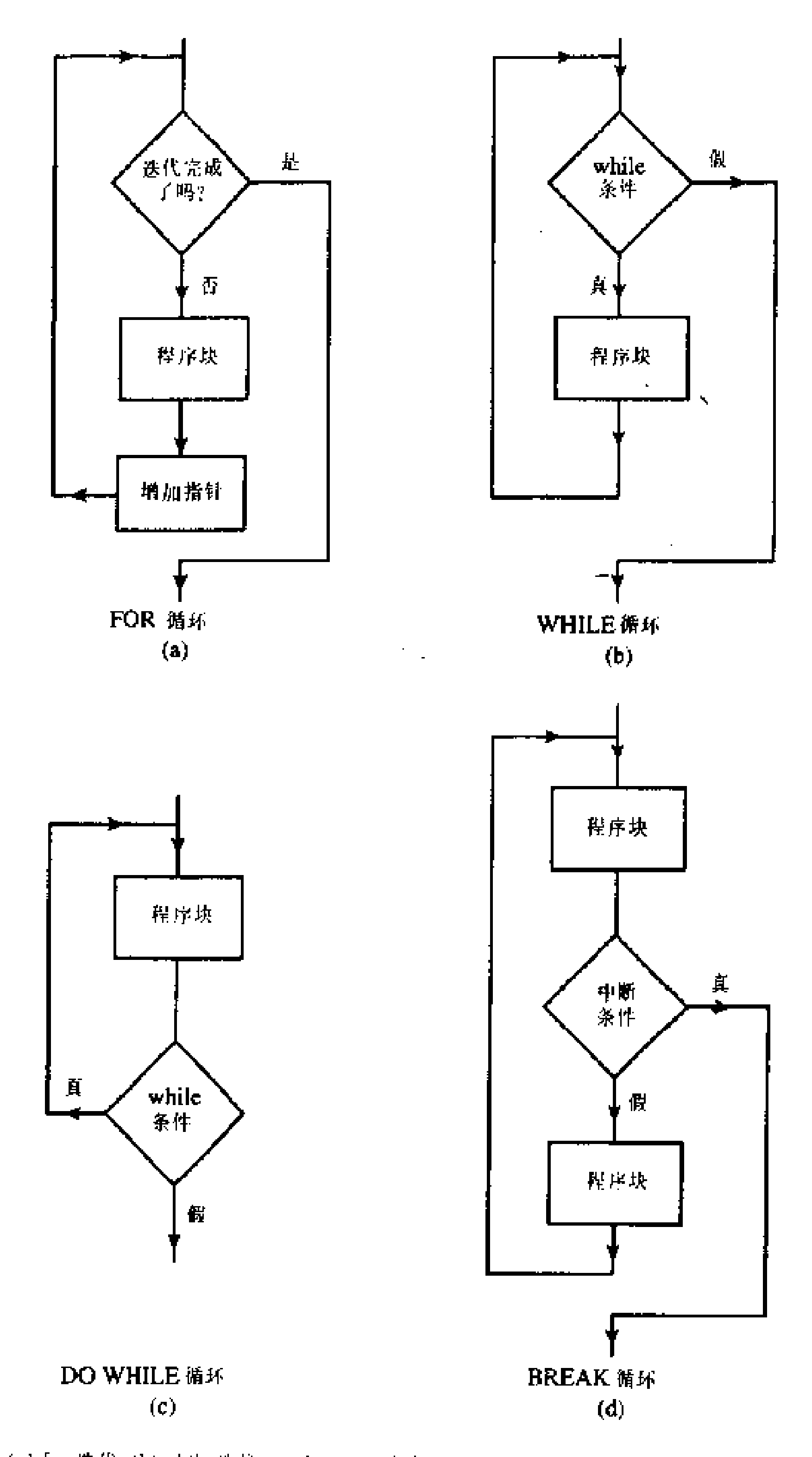
### 1.1.1 控制结构

执行程序随时间而展开，但却不是严格地按照程序书写的顺序直线展开的。影响着语句执行顺序或影响着语句是否执行的程序语句叫**控制语句**。控制语句本身从不表示任何意义，它们仅在所控制的语句组或语句块中才表达一定的意义。如果假定这些语句块是含有句子的一段文章，那么最好认为控制语句是这段文章的行首空格和句子间的标点，而不是句子中的词汇。

我们现在可以说明结构化程序设计的目的是什么了，这就是控制程序使程序看起来清晰明了。可见，这个目的与计算机如何读程序无丝毫关系。和前面所讲的一样，计算机并不理会程序员是否用了结构化程序设计，而阅读程序的人却非常关心程序结构化。一旦发现调试结构良好的程序比结构模糊的程序容易得多的时候，你自身也会关心程序结构化。

读者可以用两种互补的方法来实现结构化程序设计的目的。第一，熟悉少数基本的、在编程中经常要用到的控制结构。这些控制结构在大多数编程语言中都有方便的表示形式。除了学习这些标准控制结构(见图1.1.1)以外，读者应该在尽可能的范围内学会考虑自己的编程任务。在写程序时，应该学会习惯于用一致的、惯用方式来表示这些标准控制结构。

"这不是阻碍了创造性吗?"我们的学生有时会这样问。是的,正象奏鸣曲的形式阻碍了莫扎特的创造性,或十四行诗的韵律格式阻碍了莎士比亚的创造性一样。关键是,当创造性意味着是交流时,在格式上受到适当的限制可以更有利于进行交流。




(a) for 迭代;(b)while 迭代;(c)do while 迭代;(d)break 迭代;(e)if 结构;(f)switch 结构

图1.1.1 结构化程序设计中的标准控制结构

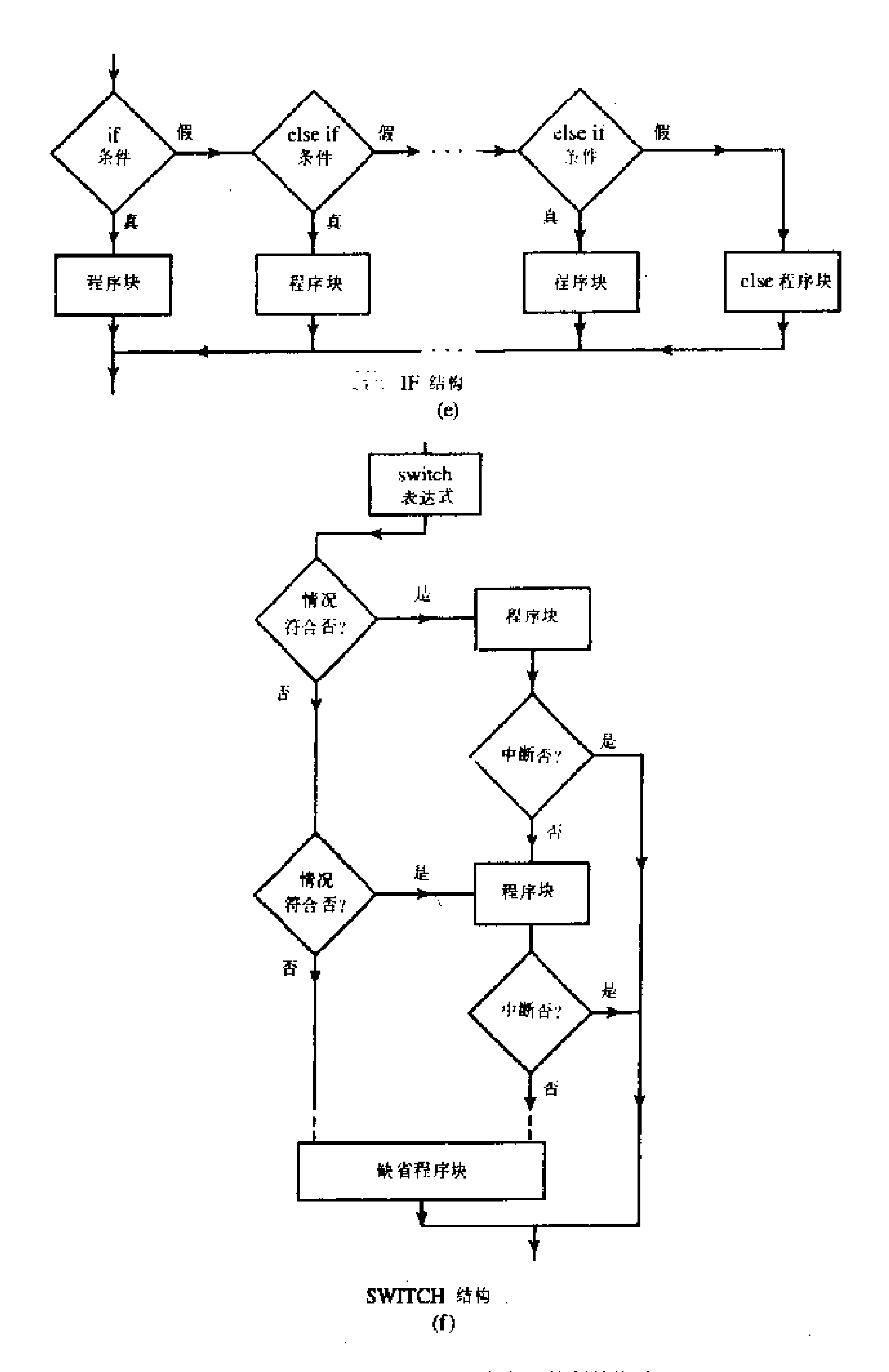



图1.1.1 结构化程序设计中的标准控制结构(续)

第三，尽可能避免使用不易一眼看出控制块或控制对象的控制语句。这意味着，在实际应用中，读者必须尽可能避免使用语句标号和goto语句。goto语句并没有太大危险(虽然它确实干扰了人们阅读程序)，而语句标号才是祸根。事实上，任何时候当你在读程序时，碰上一个语句标号，你都会条件反射地变得恐慌起来。为什么?这是因为，下面的问题会习惯性地接踵而至：指向这一标号的控制来自哪一分支?它可以来自程序中的任何地方!在什么情况下会导致这一分支转向标号?可能会是任何情况!这样使确定的事变为不确定，使对程序的理解可能会陷入沼泽中。

我们编排了一些例子，使这些考虑更为具体。

### 1.1.2 标准结构目录

**"For"迭代**　C语言中，简单的迭代是由for循环实现的，例如：

```
for (j=2; j<=1000; j++){
    b[j]=a[j-1];
    a[j-1]=j;
}
```

注意，我们总是将受控制结构作用的代码块按锯齿形排列，而结构本身不呈锯齿状。注意，我们还有一个习惯是，将起始的大括号与for语句放在同一行，而不是下一行。这样可节约整行。

**"If"结构**　C中的这个结构与Pascal、Algol、FORTRAN和其它语言中的非常相似，其典型形式如：

```
if (...){
    ...
}
else if (...){
        ...
}
else{
    ...
}
```

由于仅当程序块中有一条以上语句时，复合语句才需要大括号，C中的if结构不如FORTRAN或Pascal中的相应结构明确。在构造嵌套的if子句时一定要小心。例如：

```
if (b>3)
    if (a>3) b += 1;
else b -=1;                /* 有问题! */
```

由于有下一行缩进的语句来判别，程序员对这段程序的意图是："若b大于3且a大于3，则b加1。若b不大于3，b减1。"但是按照C语言规则，这段程序的实际意义是："若b大于3，则判断a值。若a大于3，则b增加1，若a小于或等于3，则b减1。"关键是else子句与最近打开的if子句相关连，无论把它放在页面的什么地方。如用括号来包含，可以很容易地解决这种意义混淆的问题。在一些情况下，这样做在技术上也许是多余的，但却明确了编程者的意

图并且改进了程序。因此上面的程序片段应写为：

```
if (b>3){
    if(a>3)  b   +=1;
} else {
    b   --1;
}
```

下面是一个主要含有 **if** 控制语句的工作程序：

```
#include <math.h>
#define IGREG (15+31L * (10+12L * 1582))                1582年10月15日采用格里历(阳历)

long julday (int mm, int id, int iyyy)
    在此程序中，julday 返回由年 iyyy，月 mm，日 id 确定的历书日期所对应的正午开始的凯撒历日数，所有变量都是
    整型。正的年份代表公元后(A.D.)，负的年份代表公元前(B.C.)。记住公元前1年之后是公元1年。
{
    void nrerror(char error-text[]);
    long jul;
    int ja,jy=iyyy,jm;

    if (jy== 0) nrerror("julday: there is no year zero.");
    if (jy < 0) ++j y;
    if (mm > 2) {
        jm=mm+1;                                         这是 IF 结构块的范例
    } else {
            --jy
            jm=mm+13;
    }
    jul = (long) (floor(365.25 * jy)+floor(30.6001 * jm)+id+1720995);
    if (id+31L * (mm+12L * iyyy) >= IGREG) {             测试是否转变为格里历
        ja=(int)(0.01 * jy);
        jul +=2-ja+(int) (0.25 * ja);
    }
    return jul;
}
```

(天文学家把24小时算作一个周期，起始和结束都在正午，这种整周期数就是凯撒历日数。凯撒历日的零是在很久很久以前。一个方便的参考点是2440000凯撒历日起始于1968年5月23日正午。如果知道历书中某天正午起始的凯撒历日数，那么将其加1再对7求模，就可知道该天是星期几。结果是零对应星期天，1对应星期一，…，6对应星期六。)

**"While"迭代** 除 FORTRAN 外，许多优秀的语言都提供了象如下C范例的循环结构：

```
While (n<1000)  {
        n * =2 ;
        j + =1 ;
}
```

在每次迭代之前检测控制项(这一例中是 n<1000)，是这一结构的典型特点。若控制项不为真，所包含的语句将不执行。特殊情况时，即当 n 大于或等于1000时，遇到这段程序，甚至一次都不执行。

**"Do-while"迭代** 是与 **while** 迭代成对的，在每一循环的末尾测试控制条件的控制结

构。在C中，其结构如下：

```
do {
    n *=2;
    j +=i;
} while (n<1000);
```

在这种情况下，所包含的语句至少要执行一次，与n的起始值无关。

**"Break"迭代。** 在这种情况下，程序作不定次数的循环，直到循环中某一处(可能不只在一个地方)的条件测试为真。在那一处，希望退出循环并处理随后的事情。在Pascal和标准FORTRAN中，这个结构需要用到语句标号，因此对程序的清晰有所损害。在C中，简单的break语句就可实现这种结构，它可用来中断最里层的for、while、do、或switch结构的运行，进而执行下一顺序指令。一个典型的break语句的应用是：

```
for (;;) {
        [测试前的语句]
        if (…) break;
        [测试后的语句]
    }
    [下一顺序指令]
```

下面是一个运用几个不同的迭代结构的程序。有人曾请我们其中一人为捕猎食腐动物而找一个正好是星期五、同时又是13号和满月的日期。下列程序正是完成那个任务的程序。作为副产品也附带给出了所有其它是星期五、同时又是13号的日期。

```
#include <sdio.h>
#include <math.h>
#define ZON -5.0                                    时区-5是东部是标准时间
#define IYBEG 1900                                  要搜索的日期范围
#define IYEND 2000

int main(void /* program badluk */
{
    void flmoon (int n, int nph, long *jd, float *frac);
    long juluday(int mm, int id, int fryy);
    int ic, icon, idwk, im, iyyy, n;
    float timzon = ZON/24.0,frac;
    long jd, jday;

    printf("\nFull moons on Friday the 13th from %5d to %5d\n",IYBEG,IVEND);
    for (iyyy=IYBEG; iyyy<=IYEND; iyyy++){          每年的循环
        for (im=1; im<=12;im++){                    每月的循环
            jday=julday(im,13,iyyy);                13号是星期五吗?
            idwk=(int) ((jday+1) % 7);
            if (idwk == 5) {
                n=(int)(12.37*(iyyy-1900+(im-0.5)/12.0));  n值是对从1900年以来有多少次
                icon=0;                             满月的第一次估计。把它反馈到月相程序并反复调整它,
                for (;;) {                          直到确定所期望的13日是满月。变量icon标志调整方向
                    flmoon(n,2,&jd,&frac);          得到满月的日子n
                    frac=24.0*(frac+timzon);        转变到恰当时区的小时数
                    if (frac < 0.0) {               将从正午时开始的凯撒历日转为由午夜开始的民用日
                            --jd;
                        frac += 24.0;
```

```
                }
                if (frac > 12.0) {
                    ++jd;
                    frac -= 12.0;
                }eles
                    frac += 12.0;
                if(jd == jday) {                                我们遇到目标日期了吗?
                    printf("\n%2d/13/%4d\n",im,iyyy);
                    printf("%s %5.1f %s\n","Full moon",frac,
                        "hrs after midnight (EST)");
                    break;                                      break 结构部分,匹配时情况
                }else{                                          没有到达目标日期
                    ic=(jday >= jd ? 1:-1);
                    if (ic == (-icon)) break;                   另一个 break,没有匹配的情况
                    icon=ic;
                    n += ic;
                }
            }
        }
    }
  }
  return 0;
}
```

如果好奇,那么既是13日(东部标准时间)又是星期五,同时又是满月的日期是:1903年3月13日,1905年10月13日,1919年6月13日,1922年1月13日,1970年11月13日,1987年2月13日,2000年10月13日,2019年9月13日和2049年8月13日。

**其它"标准"结构** 我们建议避免使用这些结构。每一种程序语言都有一些"诱人之物",使得设计者不禁沉醉其中。这些语言在当时看起来似乎是一些好的想法,不幸的是经不住时间的检验!它们使用户的程序难以译成其它语言,并且可读性差(因读者不熟悉这些不常用的结构)。然而几乎总是可以用其它方法来替换这些结构。

在 C 中,最有疑问的控制结构是 switch...case...default 结构(见图1.1.1)。这种结构由于不确定性会造成麻烦,而且不同的编译器,在其控制表达式中要求不同的数据类型,一般支持 char 和 int 类型。实际上,它总是能用更易理解和编译的 if...else 结构来取代。ANSI C 允许控制表达式可以为长型类数据,而许多老的编译器是不行的。通常 continue 结构总能用 if 结构替代,而不失程序的清晰性。

### 1.1.3 关于"高级论题"

用小号字体书写的内容,如本小节,标明是"高级论题"。它或者是该章主要论题之外的内容;或者是要求有更高的数学基础;或者有的是具有更深远意义的论题;或者有的是没有经过很好检验的算法。如果首次阅读本书时,跳过这些高级论题,读者不会丢失任何重要的内容。

读者可能已注意到,程序 **badluk** 在进行年月循环时,避免了使用将凯撒历日转换为历书日的任何算法。这种转换程序在结构上不是很有趣的,但偶而是有用的,所以列出如下:

```
#include <math.h>
#define IGREG 2299161

void caldat (long julian, int *mm, int *id, int *iyyy)
```

下面给出 julday 函数的反函数。这里 julian 是儒略历日数的输入，程序输出 mm，id 和 iyyy 是分别表示月、日、年，儒略历日从此天的正午算起。

```
{
    long ja,jalpha,jb,jc,jd,je;
    if (julian >= IGREG) {         跨接格里历,产生此修正
        jalpha=(long)(((float) (julian-1867216)-0.25)/36524.25);
        ja=julian+1+jalpha-(long) (0.25*jalpha);
    } else                         否则不修正
        ja=julian;
    jb=ja+1524;
    jc=(long)(6680.0+((float)(jb-2439870)-122.1)/365.25);
    jd=(long)(365*jc+(0.25*jc));
    je=(long)((jb-jd)/30.6001);
    *id=jb-jd-(long) (30.6001*je);
    *mm=je-1;
    if (*mm > 12) *mm -= 12;
    *iyyy=jc-4715;
    if (*mm > 2) --(*iyyy);
    if (*iyyy <= 0) --(*iyyy);
}
```

(适合于各种历史性日历的其他日历算法，请参阅[8])

参考文献及进一步读物：


Kernighan, B. W. 1978, *The Elements of Programming Style* (New York: McGraw-Hill). [1]

Yourdon, E. 1975, *Techniques of Program Structure and Design* (Englewood Cilffs, NJ: Prentice-Hall). [2]

Jones, R., and Stewart, I. 1987, *The Atr of C Programming* (New York: Springer-Verlag). [3]

Wirth, N. 1983, *Programming in Modula-2*, 3rd ed. (New York: Springer-Verlag). [4]

Stroustrup, B. 1986, *The C++ Progamming Language* (Reading, MA: Addison-Wesley). [5]

Borland International, Inc. 1989, *Turbo Pascal 5.5 Object Oriented Programming Guide* (Scotts Valley, CA: Borland International). [6]

Meeus, J. 1982, *Astronomical Formulae for Calculators*, 2nd ed., revised and enlarged (Richmond, VA: Willmann-Bell). [7]

Hatcher, D. A. 1984, *Quarterly Journal of the Royal Astronomical Society*, vol. 25, pp. 53-55; see also *op. cit.* 1985, vol. 26, pp. 151-155, and 1986, vol. 27, pp. 506-507. [8]


## 1.2 用 C 语言作科学计算的一些协议

C 语言最初是为系统编程工作而不是为科学计算所设计的。相对于其它高级编程语言，C 在许多方面使程序员“很贴近机器”。它的运算符丰富，能对机器语言指令集的多数功能直接寻址。它有许多不同的数据类型(长型(long)和短型(short)、有符号型(signed)和无符号型(unsigned)整数、浮点型(floating)和双精度型(double)实数、指针类型等)，以及以简洁的格式进行有效的转换。它定义了基于指针的算术运算，指针可以很巧妙地与数组地址相联系，并与许多计算机的变址寄存器结构高度兼容。

可移植性一直是 C 语言的另一大优点。C 语言是 UNIX 操作系统的基础语言。这种语言和操作系统确确实实已在上百种不同的计算机上得到了应用。语言的通用性、可移植性和灵活性已经吸引了越来越多的科学家和工程师。它通常用于实验硬件的实时控制，但是作为

控制目的的操作系统，标准的 UNIX 核还不够理想。

C 在高水平的科学计算中的应用，例如数值分析、建模、浮点数值计算工作等，一般发展较慢。一方面是因为，事实上对1960年以前出生的所有科学家和工程师，以及以后出生的大多数科学家和工程师来说，FORTRAN 已占有如同他们母语一样的牢固地位。另一方面是因为，计算机科学家很晚（我们固执地认为）才认识到计算机语言的不足之处。例如，没有好的方法可计算数值的整数幂，甚至小的整数幂以及“从浮点型(float)到双精度型(double)的隐式转换”问题等。推荐的 ANSI C 标准已克服了许多（虽然不是全部）缺陷，无疑地，另外一些不足也将随着时间的推移而消失。

另外，阻碍众多科学家转向 C 的原因是，由于编写程序时间的限制，而且明显地缺少高质量的科学或数值计算库。我们推出本版《C 语言数值算法程序大全》就是为填补这一漏缺。当然，我们并没有说它已完全解决了此问题。但是我们确实希望能激起进一步的努力，通过实例推出一套合理而实用的针对 C 语言科学计算的协议。

非常需要有一套用于 C 的编程协议。但不是为了要克服语言本身的限制（我们多次从 Pascal 中得到经验），而是为了从众多的机会中选择最好和最合乎规范的技术方法，然后将这些技术完全一致地应用于各个程序。在本节的后面部分，我们将提出一些问题，并说明本书所有程序中所采用的协议。

### 1.2.1 函数原型和标题文件

ANSI C 允许用“函数原型”来定义函数，“函数原型”标明了每个函数参数的类型。如果一个原型的函数说明或定义是可视的话，编译器就能检验一个给定函数是否用正确的变量类型进行函数调用。本书中所有程序都是采用 ANSI C 的原型形式。为了便于读者使用“传统的 K & R”C 编译器，本书的 C 语言磁盘中包括了两套完整的程序，一套用于 ANSI，另一套用于 K & R。

为了理解原型，最简单的方法就是用范例来说明。一个传统的 C 函数定义如下：

```
int g (x,y,z)
int x,y;
float z;
```

在 ANSI C 中它成为：

```
int g (int x, int y, float z)
```

而一个没有参数的函数可使参数类型空缺。

函数说明（和函数定义相比较）是用来将此函数“引入”到调用它的程序中。调用程序需要知道函数的变量个数和类型以及返回值的类型。在函数说明中，允许省略参数名。因此，上面函数的说明可写成：

```
int g(int, int, float)
```

如果一个 C 程序由许多源文件构成，编译器在没有其它辅助的情况下，不能检验每个函数调用的一致性。在这种情况下，最好的方法如下：

- 每个外部函数，在标题(.h)文件中，应有一个简单的原型说明。

• 带有函数定义的源文件也应该包括在标题文件中，以使编译器能够检验说明中的类型和定义是否一致。

• 调用函数的每一个源文件都应该包含适当的标题文件(.h)。

• 一个调用函数的程序也可以将函数的原型说明包括进去。当扩展一个程序时，这样做通常很有用。因为它把一个函数的变量类型给出可视的提示（编译器可通过公有的标题文件检验）。最后，当程序调试后，程序员可以再返回去删掉多余的内部说明。

在本书的程序中，nr.h 是包含所有原型的标题文件，它列在附录 A 中。读者应该将语句#include nr.h 放在包含本书程序的每一个源文件的顶部。因为通常在单个的源文件中，包括不止一个数值算法程序，所以我们在本书所列的单个程序前面，没有印上这个#include 说明语句。但是，读者在自己的程序中一定要包括进去。

为了备份，并且与上面所列的(•)说明中最后一项相一致，我们在本书程序中，对所调用程序的函数原型进行了内部说明（这也使得程序更易读）。唯一例外的是，在少数经常使用的应用程序（下面叙述）中，用另外的标题文件 nrutil.h 进行说明。这样，只要需要，语句# include nrutil.h 就可明确地印刷出来。

关于标题文件 nr.h，还有很重要的一点是，如同我们在磁盘中装配的那样，它同时包含 ANSI C 和传统的 K & R 类型说明。因此，要调用 ANSI 形式，必须定义以下任一宏指令：__STDC__，ANSI 或 NRANSI（最后一个名字是给读者一种引用，它不会和前两个名字的其它可能应用相冲突）。如果读者有 ANSI 编译器，就一定要用这些宏定义来调用 ANSI 形式的标题文件。典型的方法就是，在编译命令行处加一个开关，如"-DANSI"。

关于文件 nr.h 的细节，请阅读附录 A。

### 1.2.2 向量和一维数组

在 C 语言中，指针和数组具有密切而巧妙的关系。如表达式 a[j]所代表的值可定义为*((a)+(j))，即"指针 a 增加 j 后所得地址的内容"。这种定义的一个直接结果是，如果 a 指向一个合法数据位置，那么数组元素 a[0]总是被定义。C 中的数组自然地是"零起始"或"零偏移"。由语句 float b[4]；所说明的数组，它的有效元素为 b[0]、b[1]、b[2]和 b[3]，而没有 b[4]。

现在，我们需要用一种标志来指明一个数组下标的有效范围。（这个问题会在本书中反复出现！）在上例中，b 下标的范围应表示为 b[0..3]（从 Pascal 中借用来的一种表示方法）。一般地，由 float a[$M$]；说明的数组范围是 a[0..$M$-1]，若将 float 换成其它数据类型也是同样的。

问题是，许多算法很自然地喜欢从 1~$M$，而不是从 0~$M$-1。的确，读者总是能对它们进行转换，但却要为数组下标范围的转换而背上精神包袱，至少也会引起分心。因此，最好是用一致的方法。我们利用 C 语言的功能来解决这个问题。考虑：

```
float b[4], *bb;
bb=b-1;
```

指针 bb 现在指向了 b 的前一个位置。一个直接的结果是数组元素 bb[1]，bb[2]，bb[3]和 bb[4]都存在了，换言之，bb 的范围是 bb[1..4]。我们将把 bb 看作为单位偏移向量（有关这

技术的详细讨论见附录 B)。

在算法中,有时用零偏移向量方便,而有时用单位偏移向量更方便。这取决于用哪种方法解决手头的问题更合乎规律。例如,多项式 $a_0+a_1x+a_2x^2+\cdots+a_nx^n$ 的系数明显地要用零偏移向量 a[0...n],而 N 点数据向量 $x_i, i=1...N$,却应该用单位偏移 x[1..N]。当本书的程序用数组作变元时,在它的标题注释处总是给出所需下标范围。例如:

```
void someroutine (float bb[], int nn)
    这个程序是用向量 bb[1..nn]来做某些事情。
```

现在,假定用户要用函数 someroutine()在长度为 7 的向量上实现其功能。如果用户的向量名为 aa,它已经是单位偏移量(有效范围是 aa[1..7]),那么就可直接调用 someroutine(aa,7)。函数 someroutine()可能因某些逻辑上的或至少是美学上的理由,要求使用单位偏移向量。

但假如现在长度为7的向量,名为 a,偏偏用的是 C 固有的零偏移数组(范围是 a[0...6])。也许这是读者不赞同我们的美学偏见所造成的一种情况。为了能使用我们的函数,难道读者必须将 a 的内容一个一个地拷贝到另一个单位偏移的向量中吗?不必!难道还不得不定义一个新的指针 aaa,并使它与 a-1相等吗?也不必!读者只需简单引用 someroutine(a-1,7);就可以了。那么,现在从我们函数中看到的 a[1],实际上是读者程序中看到的 a[0]。换言之,有了这些说明,读者就可以使协议举一反三了。

原谅我们对这些要点作过多说明。我们是想把读者从零偏移中解脱出来,C 只是鼓励使用零偏移(在我们看来),但并不一定要求使用零偏移。最后提出的一个要点是,实用文件 nrutil.c(在附录 B 中全部列出),它包含了对任意长度的任意偏移向量的内存分配函数。这些函数的概要如下:

```
float *vector (long nl, long nh)
    对一个范围为[nl..nh]的浮点型(float)向量分配地址。

int *ivector (long nl, long nh)
    对一个范围为[nl..nh]的整型(int)向量分配地址。

unsigned char *cvector (long nl, long nh)
    对一个范围为[nl..nh]的无符号字符型(Char)向量分配地址。

unsigned long *lvector (long nl, long nh)
    对一个范围为[nl..nh]的无符号长型(long)向量分配地址。

double *dvector (long nl, long nh)
    对一个范围为[nl..nh]的双精度型(double)向量分配地址。
```

上述实用函数典型运用是,首先说明 float *b;,然后令 b=vector(1,7);,从而使范围 b[1..7]存在,并且使 b 能在任何调用单位偏移向量的函数中通过。

文件 nrutil.c 中,还包含了相应的内存释放程序:

```
void free-vector (float *v, long nl, long nh)

void free-ivector (int *v, long nl, long nh)

void free-cvector (unsigned char *v, long nl, long nh)

void free-lvector (unsigned long *v, long nl, long nh)
```

```
void free-dvector (double *v, long nl, long nh)
```

其典型运用是 free-vector (b,1,7);。

我们的库函数广泛地使用了以上实用程序,来分配和释放向量工作空间。我们也推荐读者在主程序和其它过程中运用它们。注意,如果想在 IBM-PC 兼容机上分配长度大于64k 的向量,就应在程序 nrutil.c 中所有出现 malloc 的地方,都用读者的编译器的特殊内存分配函数来替代。这种替换同样适用于下面将要讨论的矩阵分配。

### 1.2.3 矩阵和两维数组

零偏移和单位偏移在这里仍是一个要解决的问题。但让我们先把它往后放一放,以便先解决一个更基础的问题,即**变尺寸数组**(FORTRAN 术语)或**一致性数组**(Pascal 术语)。这些数组是将自身连同它们二维长度的实时信息一起传给某一函数。系统程序员很少处理二维数组,并且几乎从不使用大小是变化的、仅在运行时才知道大小的的二维数组。然而这些数组却是科学计算中经常要遇到的。很难想象仅用固定尺寸的矩阵能实施矩阵求逆的程序!

从技术上说,C 编译器没有理由不允许采用如下句法

```
void someroutine (a,m,n)
float a[m][n];          /* 非法说明 */
```

也没有理由每次进入 someroutine()时,不允许发出代码对 m 和 n(或任何的表达式)赋值,以实现数组大小的变化。很可惜,实际上 C 语言的定义却禁止采用上述程序段,而且在 C 中,要实现数组大小可变,还需一些额外的技巧。下面我们将看到一种努力的结果。

在 C 语法中关于两维数组的引用,有一句微妙的比较含糊的话。让我们来解释一下,并使它变得对我们有利。考虑对(例如)浮点型数值 a[i][j]的数组引用,这里的 i 和 j 定为整型数。对这个引用,C 编译器将产生完全不同的机器码,这取决于标识符 a 是如何说明的。若说明 a 为固定大小的数组,例如 float a[5][9];,则机器码是:"对地址 a 加上9乘 i,然后再加 j,返回它所定位的地址中的数值"。注意,需要知道常数9,它用来作一个整型乘法运算(见图1.2.1)。

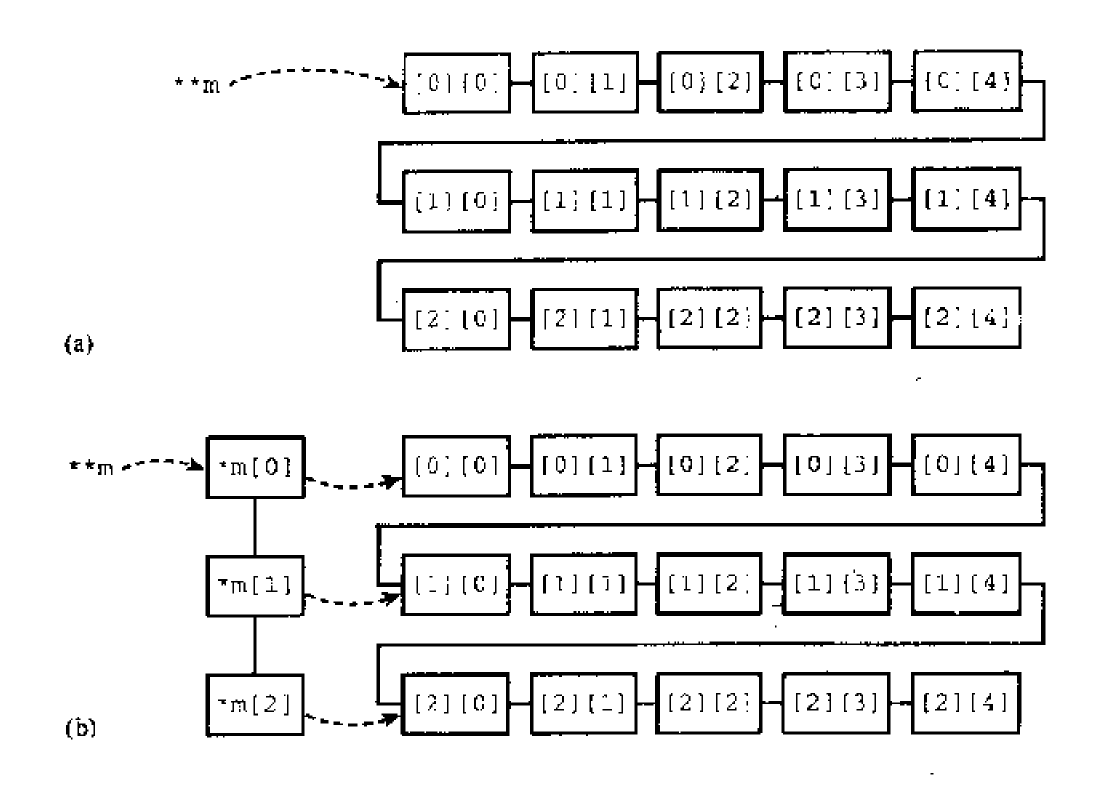
另一种情况,若说明 a 为 float **a;,则 a[i][j]的机器码是:"对 a 的地址加 i,取此地址中所得值为新的地址,再对其加 j,然后返回这个新地址中的值"。注意,a[][]的基本尺寸完全没有参与运算,而且无乘法运算,只是用所增加的迂回代替了它。因此,不失一般性,我们具有了较前者更快、更多功用的方案。我们所花费的代价是,需要存储一个指针数组(指向 a[][]的行),并且增加了一点不便——当说明一个数组时,要记住初始化那些指针。

我们的最终目的是:避开 C 的固定大小的二维数组。因为这种数据结构不适合代表科学计算中的矩阵。我们采用"指向指针数组的指针"这个约定。指针数组的元素指向矩阵每一行的第一个元素。图1.2.1将放弃的和采用的方案作了对比。

下面的程序段显示了如何将一个固定大小为13×9的数组 a,转换成一个"指向指针数组的指针"aa:

```
float a[13][9], **aa;
int i;
aa=(float **) malloc((unsigned) 13*sizeof (float *));
```

```
for (i=0; i<=12; i++) aa[i]=a[i];        a[i]是指向a[i][0]的指针
```




点划线指出了地址起始位置，而实线连接了连续存贮单元。

(a)指向固定的两维数组的指针；(b)指向行指针数组的指针，这是本书采用的方案

图1.2.1　矩阵m的两种存储方案

标识符aa现在是一个具有下标范围为aa[0..12][0..8]的矩阵。我们可以自由地运用或修改它的元素。更重要的是，可以以它的名字aa作为一个变量传递给任一个函数。用相应的哑元变量为float　* *aa;说明的函数能够以aa[i][j]的形式对其元素寻址，而无需知其物理尺寸。

读者可能并不希望象上面程序段那样散乱地堆放自己的程序代码。而且还有一个重要问题就是，如何处理"单位偏移"的下标，而使(例如)上面的矩阵aa可在范围a[1..13][1..9]寻址。这两个问题都可由nrutil.c(附录B)中的附加实用程序解决，它们可分配和释放任意范围的矩阵。其摘要如下：

float * * matrix (long nrl, long nrh, long ncl, long nch)
　　分配一个范围为[nrl..nrh][ncl..nch]的浮点型矩阵.

double * * dmatrix (long nrl, long nrh, long ncl, long nch)
　　分配一个范围为[nrl..nrh][ncl..nch]的双精度矩阵。

int * * imatrix (long nrl, long nrh, long ncl, long nch)
　　分配一个范围为[nrl..nrh][ncl..nch]的整型矩阵。

void free-matrix(float * * m, long nrl, long nrh, long ncl, long nch)
　　释放一个由matrix分配的矩阵。

void free-dmatrix (double * * m, long nrl, long nrh, long ncl, long nch)

释放一个由 dmatrix 分配的矩阵。

```
void free-imatrix (int * * m, long nrl, long nrh, long ncl, long nch)
```
释放一个由 imatrix 分配的矩阵。

一个典型运用是：

```
float * * a ;
a=matrix (1,13,1,9);
...
a[3][5]=...
...+a[2][9]/3.0...
someroutine (a,...);
...
free_matrix (a,1,13,1,9);
```

《C 语言数值算法程序大全》中的所有矩阵都是按如上面范例处理的，我们也将它推荐给读者。

nrutil. c 中还含有另外一些处理矩阵的实用程序：第一个是函数 submatrix ()。如果需要的话，它可用来对应于一个已存在的矩阵(或它的子块)，建立一个有新偏移量的新指针。其典型应用是：

```
float * *submatrix (float * * a, long oldrl, long oldrh,
    long oldcl, long oldch, long newrl, long newcl)
```
对已存在的范围为 a[oldrl..oldrh][oldcl..oldch] 的矩阵，建立一个新指针，其范围是为[newrl..newrl+(oldrh−oldrl)][newcl..newcl+(oldch−oldcl)]的子矩阵。

这里 oldrl 和 oldrh 分别为原矩阵的低位行和高位行下标，oldcl 和 oldch 是相应的列下标，newrl 和 newcl 分别是新矩阵的低位行和低位列下标(我们不需要高位行和高位列的下标，因它们可由已给出的数据推算得到)。

两个应用的例子是：第一例，选取已知矩阵的某些内部范围，例如选取 a[4..5][2..3]为一个2×2新矩阵 b[1..2][1..2]，

```
float * *a, * *b;
a=matrix (1,13,1,9);
...
b=submatrix (a,4,5,2,3,1,1);
```

第二例，把一个已存在的矩阵 a[1..13][1..9]映射为一个新矩阵 b[0..12][0..8]。

```
float * *a, * *b;
a=matrix (1,13,1,9);
...
b=submatrix (a,1,13,1,9,0,0);
```

附带地，还可将 submatrix()用于非 float 类型矩阵，只要简单地强制它的第一个参数类型为 float * *，而将其结果强制为期望的类型，例如 int * *。

函数

```
void free-submatrix (float * * b, long nrl, long nrh, long ncl, long nch)
```

释放由 submatrix()分配的行指针数组。注意，它并不释放分配给了矩阵中数据的存储单元，因为那些空间仍在原来矩阵分配的存储空间中。

最后，如果有一个用 a[nrow][ncol]说明的标准 C 矩阵，若想把它转变成一个用指向行指针数组的指针所说明的矩阵，则可以采用如下的函数：

float * * convert_matrix (float * a, long nrl, long nrh, long ncl, long nch)
　　分配一个浮点数矩阵 m[nrl..nrh][ncl..nch]，它指向用标准 C 说明的矩阵 a[nrow][ncol]，其中 nrow＝nrh－nrl＋1，ncol－nch－ncl＋1。被调用时，第一个变元的地址应是 & a[0][0]。

(当需要利用 C 的初始化语法对一个矩阵赋值时，则可以利用这个函数。然后，便可以把这个矩阵传给本书中的程序)。下面的函数

void float free_convert_matrix (float * * b, long nrl, long nrh, long ncl, long nch)
　　释放由 convert_matrix()分配的一个矩阵。

释放被分配的矩阵，而不影响原始矩阵 **a**。

本书中，分配三维数组采用指针-指向-指针-指向-指针的结构，具体范例可以在第12.5节中的程序 **rlft3**和第17.4节中的程序 **sfroid** 中找到。必要的分配和释放函数是

float * * * f3tensor (long nrl, long nrh, long ncl, long nch, long ndl, long ndh )
　　分配一个下标范围为[nrl..nrh][ncl..nch][ndl..ndh]的浮点型三维数组。

void free-f3tensor (float * * * t, long nrl, long nrh, long ncl, long nch, long ndl, long ndh)
　　释放一个由 f3tensor()分配的浮点型三维数组。

### 1.2.4 复数运算

C 没有复数数据类型，也没有预定义复数的算术运算。但可以很容易地用文件 complex.c 中的函数(全部印刷在本书附录 C 中)来填补这个漏洞，其概要如下：

typedef struct FCOMPLEX {float r, i;} fcomplex;

fcomplex Cadd (fcomplex a, fcomplex b)
　　返回两个复数的复数和。

fcomplex Csub (fcomplex a, fcomplex b)
　　返回两个复数的复数差。

fcomplex Cmul(fcomplex a, fcomplex b)
　　返回两个复数的复数积。

fcomplex Cdiv (fcomplex a, fcomplex b)
　　返回两个复数的复数商。

fcomplex Csqrt (fcomplex z)
　　返回一个复数的复数平方根。

fcomplex Conjg (fcomplex z)
　　返回一个复数的复共轭值。

float Cabs (fcomplex z)
　　返回一个复数的绝对值(模)。

fcomplex Complex (float re, float im)

返回一个特定实部和虚部的复数。

fcomplex RCmul (float x, fcomplex a)
　　返回一个实数和一个复数的复数积。

在浮点运算中实现几个这些复数操作，不是完全无足轻重的，见第5.4节。

本书中约有6个程序对这些复数型算术函数作了很好的应用。但最终的代码并不象所想的那样易于阅读，因为熟悉的运算符+－*/被函数调用所取代。C语言的扩充C++允许运算符重定义，这将使程序具有更高的可读性。但在本书中，我们遵守标准C的规定。

我们应该提出，以上函数是假定具有以数值传送，返回和分配象FCOMPLEX这样结构(或定义为结构的fcomplex类型)的功能。所有近期的C编译器都具备这样的功能，但最初的K&R C定义中并不具备此。若读者的编译器中没有此功能，必须重新编写complex.c中的函数，使它们能传送和返回指向fcomplex类型变量的指针而不是变量本身。同样地，将修改使用这些函数的程序。

一些其它程序(如傅里叶变换**four1**和**fourn**)用"手算"对复数进行算术运算，即将实部和虚部分别作为浮点型变量。这比用complex.c中的函数能产生更高效率的程序。但代码可读性更差。对C中的复数运算问题尚无理想的解决方法。

### 1.2.5 浮点到双精度的隐式转换

在传统C中，在任何操作之前，包括算术运算和向函数传送变量，浮点型(float)变量都将自动转为双精度型。因此，所有的算术运算都是在双精度下进行的。如果一个浮点变量接受了这样的算术运算结果，高精度立即丢失。一个必然的结果是，所有实数型标准C的库函数都是双精度型的，并作双精度计算。

制定这些转换规则的正当理由是，"一点额外的精度就不会产生任何错误"和"用这种办法，函数库中每个函数仅需一种版本形式"。然而，在科学计算上，人们不需要很多经验就能认识到，这种隐式转换规则事实上完全是荒谬的！在效果上，它使我们不可能写出高效率的数值计算程序。对丢失30%或50%的速度是否值得优虑持不同的观点，这正是计算机科学家与"正规的"科学家和工程师之间一种文化上的障碍。在许多实时或工艺科学的应用中，这样的损失将导致巨大的灾难。一个应用科学家总是在用昨日的计算机试图去解决明天的问题，而计算机科学家，我们认为，总是用另外的方法。

令人欣慰的是，所推荐的ANSI C标准不允许算术运算的隐式转换，但对函数变量却一定需要这种转换，除非函数是完全原型函数，即它是由本节前面所述的ANSI说明组成。这正是我们为什么要推荐ANSI原型结构的理由，同时这也是读者需使用ANSI兼容的编译器的理由。

一些老的C编译器提供了一种可选择的编译模式，其中抑制了浮点型(float)到双精度型(double)的隐式转换。若读者可能，请用这种编译模式。在本书中，当我们写float时，意指浮点型；当我们写double时，意指双精度型，这是由于算法上的原因要求更高的精度。我们所有的程序都能默许传统的隐式转换规则，但如果没有它们，效率将更高。当然，如果实际应用需要双精度，则可以很容易地将说明从float变为double(硬性方法是，加上一条预处理语句 #define float double)。

## 1.2.6 一些技巧

我们喜欢使程序代码紧凑，避免不必要的空间，除非它们的加入能明显地使程序清晰。我们通常不在赋值运算符“=”两旁设空格。但由于历史的扭曲，一些C编译器将(不存在的)运算符“=-”认为等价于减法赋值运算符“-=”，以及将“=*”等同于乘法赋值运算符“*=”。这就是为什么在程序中，我们写例如 y= -10.0；或 y=(-10.0)；，和 y= *a；或 y=(*a)；等这种形式。

我们对不可缺少的圆括弧持同样的态度。除非已记住C操作符的运算顺序和相关规则，否则不可能有效地编写(或阅读)C程序。请熟读表1.2.6.1。

**表1.2.6.1　C操作符的运算顺序及相关规则**

| 操作符 | 含义 | 结合规则 |
|---|---|---|
| ()<br>[]<br>.<br>-> | 函数调用<br>数组元素引用<br>结构或联合元素引用<br>结构指针引用 | 从左到右 |
| !<br>~<br>-<br>++<br>--<br>&<br>*<br>(type)<br>sizeof | 逻辑非<br>位操作补<br>一元减法<br>增量<br>减量<br>地址<br>引dsl所指的内容<br>类型强制(转换)<br>字节长度 | 从右到左 |
| *<br>/<br>% | 乘<br>除<br>求余 | 从左到右 |
| +<br>- | 加<br>减 | 从左到右 |
| <<<br>>> | 位左移<br>位右移 | 从左到右 |
| <<br>><br><=<br>>= | 算术小于<br>算术大于<br>算术小于或等于<br>算术大于或等于 | 从左到右 |
| ==<br>!= | 算术等于<br>算术不等于 | 从左到右 |
| & | 位操作与 | 从左到右 |
| ^ | 位操作异或 | 从左到右 |
| \| | 位操作或 | 从左到右 |
| && | 逻辑与 | 从左到右 |
| \|\| | 逻辑或 | 从左到右 |
| ? : | 条件表达式 | 从右到左 |
| =<br>还有 += -= *= /= %=<br><<= >>= &= ^= \|= | 赋值操作符 | 从右到左 |
| , | 顺序表示(逗号运算符) | 从左到右 |

我们从不使用寄存器存储分类符。良好的经过优化的编译器，对决定将什么放在寄存器中是非常在行，而且有时最好的选择很不直观。

对于几个文件中相同外部名的定义说明和引用说明，不同的编译器采用不同的鉴别方法。我们采用最普通的方案，这也是 ANSI 标准。存储类 extern 明显地包括在所有的高级引用说明中。因此，每个外部变量的单个定义说明中省略存储类。我们在此评论这些说明的目的是，如果读者的编译器采用不同的方案，则可以自己改变程序代码。不同的编译方案在参考书[1]的 § 4.8中讨论。

我们已从侧面提到了计算数值的整数幂，尤其是平方和立方的计算问题。C 中这种操作的遗漏，也许是这种语言对科学计算程序员的最伤感情的损害。所有好的 FORTRAN 编译器都能识别象(A+B)＊＊4这样的表达式，并产生一行代码。在这一例中，仅需一次加法和两次乘法。所识别的常整数次幂可达到12，是件很典型的事。

但在 C 中，仅仅平方问题就很难！有时"宏"操作为：

```
#define  SQR(a) ((a)*(a))
```

但是，这对 SQR(sin(x))，将导致两次调用正弦函数！为避免这种情况，我们写为：

```
static float sqrarg;
#define SQR(a) (sqrarg=(a), sqrarg*sqrarg)
```

全局变量 sqrarg 现在覆盖了(并需要保持)模块的整个范围。这就有一定的危险性。另外，还需要一个与整型数平方表达式完全不同的宏。更严重的是，如果在单个表达式中有两个 SQR 运算，这种宏将会失败。因为在 C 中，表达式的求值顺序是按编译器的方向。那么，一个 SQR 估计值 sqrarg 就可能来自于同一表达式中其它的估计值，从而产生毫无意义的结果。当需要一个保证正确的 SQR 宏，则可使用下面的程序。此程序在条件表达式下，充分利用了子表达式中已被保证的充分估值。

```
static      float sqrarg;
#define     SQR(a) ((sqrarg=(a))==0.0 ? 0.0 : sqrarg*sqrarg)
```

在文件 nrutil.h 中，还包括了其它一些简单的宏运算(见附录 B)。书中许多程序都使用这些宏运算，其概要如下：

| | |
|---|---|
| SQR(a) | 浮点数平方 |
| DSQR(a) | 双精度数平方 |
| FMAX(a,b) | 求两个浮点数的最大值 |
| FMIN(a,b) | 求两个浮点数的最小值 |
| DMAX(a,b) | 求两个双精度数的最大值 |
| DMIN(a,b) | 求两个双精度数的最小值 |
| IMAX(a,b) | 求两个整型数的最大值 |
| IMIN(a,b) | 求两个整型数的最小值 |
| LMAX(a,b) | 求两个长整型数的最大值 |
| LMIN(a,b) | 求两个长整型数的最小值 |
| SIGN(a,b) | 对 a 取 b 的符号 |

也许在某一天，用C作科学编程会成为一座玫瑰花坛，但现在还需留心刺！

参考文献和进一步读物：

Harbison, S. P., and Steele, G. L., Jr. 1991, *C: A Reference Manual*, 3rd ed. (Englewood Cliffs, NJ: Prentice-Hall). [1]

## 1.3 误差、准确性和稳定性

虽然，我们假定在正规的数值分析方面，读者没有前期的训练，但对少数关键的概念必须应有一般性的了解。我们将在下面对这些内容作简要的定义。

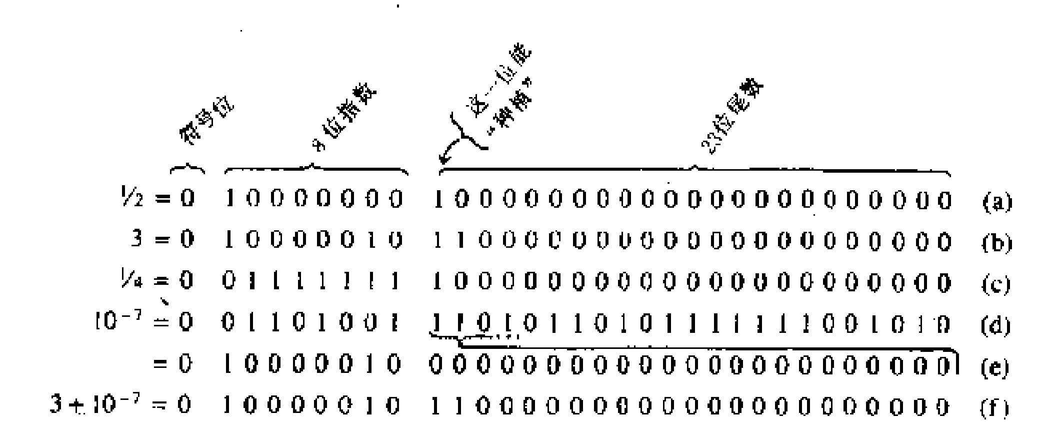
计算机不是以无限精度来存储数值的，而是以一定数量的**位**（二进制位）或**字节**（8位一组）的一种近似方式来存储数值的。几乎所有的计算机，都允许程序员在几种不同的位或字节的表示或数据类型中进行选择。数据类型可以根据所用的位数（字长）来分类；也可从更基本的方面，存储的数据是以**定点**（int 或 long）还是以**浮点**（float 或 double）格式来进行分类。

用整型表示的数是精确的。只要满足条件(i)结果不超出整型数（通常指带符号整型数）所能代表的范围，和(ii)除法看作仅能产生整数结果，舍弃余数，则整型数据之间的算术运算也是精确的。

在浮点表示中，一个数的内部表示需要一个符号位 $S$（代表加号或减号），一个精确的整指数 $e$ 和一个精确的正整尾数 $M$。这些组合在一起代表的数为

$$S \times M \times B^{e-E} \tag{1.3.1}$$

其中 $B$ 是表达式的基数（通常 $B=2$，但有时 $B=16$），$E$ 是指数的偏移（对任何给定的机器和表达式，它是一个固定的整常数）。图1.3.1显示了一个示例。




(a)数1/2（注意指数的基）；(b)数3；(c)数1/4；(d)数$10^{-7}$的机器准确表示；(e)同样的数$10^{-7}$，但作了移位使得它与3有相同的指数位，数$10^{-7}$作了移位以后，所有的有效位都损失了，变成了零，移位使得两个数有相同的指数项后，这两个数才能相加；(f)两数之和$3+10^{-7}$，在机器准确度范围内等于3。即使$10^{-7}$本身可以准确地表示，但它也不能准确地加到一个大得多的数上去。

图1.3.1 浮点数的典型32位（4个字节）表示

几种浮点位模式都能代表相同的数。例如，如果 $B=2$，前导（高阶）位是零的尾数可以左

移，即可以乘上2的幂，只要指数项减去一个补偿量就行了。“尽可能向左移位”的位模式称为**标准化**模式。大多数计算机总是产生标准化结果，这样可以不浪费尾数的任何位而因此达到较高精度的表示。由于正确的标准化尾数（当 $B=2$）的高阶位总是为1，大多数计算机根本不存储这一位而给出一位额外的精度。

浮点形式表示的数之间的算术运算是不精确的，即使运算的数能被精确地表示（即能用式(1.3.1)形式表示的精确值）。例如，两个浮点数相加需要将较小（在数值上）数的尾数首先向右移（除以2），同时相应地增大它的指数，直到两个操作数有相同的指数项。通过如上的移位，较小的数将丢掉了低阶（最小有效）位。如果两个运算数在数值上相差太大，那么较小的数将被零取代，因为它向右移而有效位都被淹没掉。

若一个最小的浮点数（在数值上），当它加上浮点数1.0后，产生一个不同于1.0的浮点数，则称这最小浮点数为机器精度 $\epsilon_m$。具有 $B=2$ 和32位字长的典型计算机的 $\epsilon_m$ 大约是 $3\times10^{-8}$。（在20章第一节中还将详细讨论机器的精度问题，并且用程序来确定它们）。粗略地说，机器精度 $\epsilon_m$ 是浮点数表示的相对精度，对应于尾数的最小精度位的变化。要认识到浮点数之间的大多数算术运算都将导致额外的至少为 $\epsilon_m$ 的相对误差。这种误差称为**舍入误差**。

$\epsilon_m$ 不是一个机器所能表示的最小浮点数，明白这一点很重要。最小浮点数取决于指数项有多少位，而 $\epsilon_m$ 取决于尾数有多少位。

舍入误差随着计算量的增加而积累起来。如果在获取运算结果的过程中，执行了 $N$ 次同样的算术运算，只要舍入误差是随机地上下出现，可能会很幸运仅产生 $\sqrt{N}\epsilon_m$ 阶总的舍入误差（其中平方根产生于随机游动）。但这个估计值可能由于下面两个原因而大大偏离：

(1)经常会发生这种情况，所用的计算规则或计算机的特点造成舍入误差优先在一个方向上积累。这时总舍入误差将是 $N\epsilon_m$ 数量级。

(2)一些特殊的不幸事件发生，使在一次运算中大大增加了舍入误差。一般来说，这种情况可追溯到两个非常接近的数相减，所得结果的有效位只是两个操作数不同的那些（极少几位）低阶位。读者可能认为这样“巧合”的减法不可能发生。但事实并不总是这样。一些数学表达式大大增加了这种情况发生的可能性。例如，在熟悉的求二次方程根的公式中：

$$x=\frac{-b+\sqrt{b^2-4ac}}{2a} \tag{1.3.2}$$

当 $ac\ll b^2$ 时，加法变得很微妙并且易于发生舍入误差。（在第5.5节中，我们将学会如何避免在这种特殊情况下的问题。）

舍入误差是计算机硬件的特性。还有另一种不同的误差，反映的是所用程序或算法的特性，与程序运行所依赖的硬件无关。许多数值算法都是计算“连续量”的“离散”近似值。例如，数值求积分就是通过计算在一组离散点集上的函数值，而不是计算“每一”点的值来进行的。或者，一个函数可由它的无限级数的有限项之和来近似计算，而不是无限项。在这些情况中，有一个可调整的参数，例如点数或项数，仅当参数趋于无穷时才能得到“真值”。任何实际运算都是参数选择为有限的而又足够大的值。

真值与实际运算所得值之差称为**截断误差**。甚至在一个假设“完美的”、有无限精度表示的并且无舍入误差的计算机上，截断误差依然存在。作为一般的规律，程序员除了选择不会增大舍入误差的算法以外，他对舍入误差总是无可奈何（见下面的“稳定性”讨论）。然而，截断误差却完全在程序员的控制之中。事实上，可以略微夸张地说，灵活地使截断误差减到最

小、实际上就是数值分析领域的全部内容！

大多数时候，截断误差和舍入误差彼此不会强烈地相互作用。计算误差可以想象为，在一个无限精度的计算机上运行所产生的截断误差，"加上"与执行的运算次数相关的舍入误差。

但有时，一个在其它方面很吸引人的方法可能是不稳定的。这意味着，在开始阶段"混入"计算的舍入误差被连续地扩大直到淹没真值。一个不稳定的算法在一个假设完美的计算机上是有用的。但在这个不完美的世界上，我们要求算法是要稳定的——或者极其谨慎地使用不稳定的算法。

这里是一个简单的，某些人为的不稳定算法例子：假设要计算所谓"黄金分割"的所有整数幂，黄金分割数为：

$$\varphi = \frac{\sqrt{5}-1}{2} \approx 0.61803398 \tag{1.3.3}$$

结果表明（可以很容易地验证）幂 $\varphi^n$ 满足简单的递推关系

$$\varphi^{n+1} = \varphi^{n-1} - \varphi^n \tag{1.3.4}$$

这样，知道了前面两个值 $\varphi^0$ 和 $\varphi^1=0.61803398$，我们可以连续地用式(1.3.4)，在每一步仅作一次减法，而不是用 $\varphi$ 作一次较慢的乘法。

不幸的是，递推式(1.3.4)还有另外一个解，即值为 $-\frac{1}{2}(\sqrt{5}-1)$。由于递推式是线性的，并且这个不期望的解在数值上比1大，因此任何由舍入误差引起的小误差都将呈指数增加。在典型的32位字长计算机上，式(1.3.4)在大约 n=16时就会给出完全错误的结果，这时幂 $\varphi^n$ 仅降到 $10^{-4}$。所以递推式(1.3.4)是不稳定的，不能用于所指定的目的。

在本书的较后部分，我们将遇到许多外形上更为复杂的稳定性问题。

# 第二章　线性代数方程组求解

## 2.0　引言

线性代数方程组的形式如下：

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1N}x_N &= b_1 \\
a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2N}x_N &= b_2 \\
a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \cdots + a_{3N}x_N &= b_3 \\
\cdots \qquad & \qquad \cdots \\
a_{M1}x_1 + a_{M2}x_2 + a_{M3}x_3 + \cdots + a_{MN}x_N &= b_M
\end{aligned}
\tag{2.0.1}
$$

其中 $N$ 个未知数 $x_j$，$j=1,2,\cdots,N$ 与 $M$ 个方程相联系。系数 $a_{ij}$ 为已知数，$i=1,2,\cdots,M$；$j=1,2,\cdots,N$。右端项 $b_i$，$i=1,2,\cdots,M$ 也是已知量。

### 2.0.1　非奇异与奇异方程组

如果 $N=M$，方程的个数与未知数个数相同，这样就有可能求得 $x_j$ 的唯一解集。分析可知，如果这 $M$ 个方程中的一个或几个方程是其它方程的线性组合，这种情况被称为**行退化**；或者如果所有的方程中某些变量是其它变量的同一线性组合，称之为**列退化**(对方阵来说，行退化意味着列退化，反之亦然)，这两种情况都得不到唯一的解。退化的方程组称为**奇异**的。我们将在第2.6节中考虑奇异矩阵的细节问题。

至少有两个附加的因素会导致数值上的错误：

· 尽管有些方程并非彼此精确的线性组合，但它们可能很接近线性依赖，而在求解过程的某个阶段，机器的舍入误差使它们成为线性依赖的了。这时计算过程将失败，并会提示求解已失败。

· 求解过程中，舍入误差的积累也会使结果与真实解之间相差甚远。这个问题在 $N$ 很大时特别容易发生，但计算程序在算法上并无错误。然而将结果直接代回原来的方程就会发现，由它计算出来的 $x$ 的解集却是错误的。由于求解过程中会不断发生相近抵消，方程组越接近奇异的，这种情况越易发生。事实上，所计算的项可视为有效数字不幸被全部舍去的特殊情况。

复杂的线性方程求解程序包中，用了许多复杂的方法来检测和(或者)改正这两种会引起错误的情况。当求解大的线性方程组时，就会觉得需要这样的复杂处理。由于不存在所谓典型的线性问题，所以很难给出任何固定的准则。这里给出一个粗略的建议：$N$ 在 20 至 50 之间的线性方程组，如果方程组不是接近奇异的，可以按惯例用单精度求解(32 位浮点表示)，而不必借助于更复杂的方法。如果用双精度(60 或 64)，$N$ 可以扩大到几百，在这

点上，限制因素几乎总是机器时间，而不是精度。

甚至对于 $N$ 上千的更大的线性方程组，当系数是稀疏的(即大部分为零)时也可以求解，可以采用利用了稀疏性的特点的方法求解。我们将在第2.7节中进一步讨论。

另一方面在谱问题上，我们经常遇到的线性问题，由于其内在的属性是接近奇异的。在这种情况下，即便对于 $N=10$ 的情况(然而极少存在对 $N=5$ 的情况)，可能都需要借助于很复杂的方法。奇异值分解(第2.6节)技术，有时能把奇异的问题变成非奇异的问题，这时额外的复杂处理就没有必要了。

## 2.0.2 矩阵

方程组(2.0.1)可写成下面的矩阵形式

$$\mathbf{A}\cdot\mathbf{x}=\mathbf{b} \tag{2.0.2}$$

式中黑点代表矩阵相乘，$\mathbf{A}$ 为系数矩阵，$\mathbf{b}$ 为写成列向量形式的右端项，

$$\mathbf{A}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ & & \cdots & \\ a_{M1} & a_{M2} & \cdots & a_{MN} \end{bmatrix} \qquad \mathbf{b}=\begin{bmatrix} b_1 \\ b_2 \\ \cdots \\ b_M \end{bmatrix} \tag{2.0.3}$$

习惯上，元素 $a_{ij}$ 的第一个下标表示行，第二个下标表示列。大多数实际应用中，不必知道矩阵在计算机存储器中是如何存放的，可以简单地用二维地址引用矩阵元素，例如 $a_{34}$ = a[3][4]。在第1.2节中我们已经看到这个 C 语言符号的背后，事实上隐藏了一个相当精巧而又通用的物理存储方案："指向行指针数组的指针"。这方面的内容可以复习一下有关的章节。有时候了解一下内情是有帮助的，例如要想通过 a[i]来遍历整行 a[i][j]，j=1,2,…,N。

## 2.0.3 计算线性代数的任务

我们来考虑一下，下面这些任务作为进入本章要讨论内容的开始：

- 矩阵方程 $\mathbf{A}\cdot\mathbf{x}=\mathbf{b}$ 的求解。其中 $\mathbf{x}$ 为未知向量，$\mathbf{A}$ 为系数方阵，中间的圆点意味着矩阵相乘，$\mathbf{b}$ 为已知右端向量(第2.1节-第2.10节)。

- 不止一个的矩阵方程 $\mathbf{A}\cdot\mathbf{x}_j=\mathbf{b}_j$ 的求解问题。式中 $\mathbf{x}_j$ 为未知向量的集合，$j=1,2,\cdots$，每一个 $\mathbf{x}_j$ 对应一个不同的已知右端项向量 $\mathbf{b}_j$。这项任务中，关键的简化是矩阵 $\mathbf{A}$ 为常量矩阵，而右端项 $\mathbf{b}$ 为变化的(第2.1节-第2.10节)。

- 矩阵 $\mathbf{A}^{-1}$ 的计算。$\mathbf{A}^{-1}$ 为方阵 $\mathbf{A}$ 的逆，即 $\mathbf{A}\cdot\mathbf{A}^{-1}=\mathbf{A}^{-1}\cdot\mathbf{A}=\mathbf{1}$，$\mathbf{1}$ 表示单位矩阵(除对角线上的元素是1外，其余元素均为零)。对一个 $N\times N$ 矩阵 $\mathbf{A}$ 来说，这项任务与前一项任务在 $N$ 个不同 $\mathbf{b}_j(j=1,2,\cdots N)$ 为单位向量($\mathbf{b}_j$ 除第 $j$ 个元素为1外均为零元素)的情况下是等价的。相应的 $\mathbf{x}$ 为 $\mathbf{A}$ 的逆矩阵的列(第2.1节和第2.3节)。

- 方阵 $\mathbf{A}$ 的行列式的计算(第2.3节)

如果 $M<N$，或者 $M=N$ 时，方程组是退化的，则有效方程的个数少于未知数个数。在这种情况下，或者无解，或者有不止一个的解向量 $\mathbf{x}$。对后者，解空间包括一个特殊解 $\mathbf{x}_p$，把它加在(不失一般性)$N-M$ 个向量(称之为矩阵 $\mathbf{A}$ 的零空间)的任意线性组合上。寻找 $\mathbf{A}$ 的

解空间这项任务包括

- 矩阵 $\mathbf{A}$ 的奇异值分解

这个专题在第2.6节中涉及。

相反的情况是方程个数多于未知数数目，即 $M>N$。这种情况发生时，通常对方程组(2.0.1)没有解向量 $\mathbf{x}$，这个方程组被称作**超限定的**。然而这种情况会经常出现，最好的解决方案是，寻找一个能同时近似满足所有方程的“折衷”解。如果近似程度被限定在最小二乘意义上，也就是说，方程(2.0.1)左端和右端之差的平方和最小，于是此超限定的线性问题归并为一个(通常是)可解的线性问题，称为

- 线性最小二乘问题

归并后，待求解的方程组可写成 $N\times N$ 方程组

$$(\mathbf{A}^T\cdot\mathbf{A})\cdot\mathbf{x}=(\mathbf{A}^T\cdot\mathbf{b}) \tag{2.0.4}$$

式中 $\mathbf{A}^T$ 表示矩阵 $\mathbf{A}$ 的转置。方程组(2.0.4)称为线性最小二乘问题的**正规方程组**。奇异值分解与线性最小二乘问题之间有着紧密的联系，后者也将在第2.6节中讨论。顺便提醒一下，直接求正规方程组(2.0.4)的解并非总是求解最小二乘问题的最好方法。

本章中的其它一些论题包括

- 解的迭代改进(第2.5节)
- 各种特殊形式的矩阵：对称正定矩阵(第2.9节)，三对角矩阵(第2.4节)，带状对角矩阵(第2.4节)，托普雷兹矩阵(Toeplitz)(第2.8节)，范德蒙矩阵(第2.8节)，稀疏矩阵(第2.7节)
- 斯特拉森(Strassen)的“快速矩阵求逆”(第2.11节)。

## 2.0.4 标准子程序包

我们不能指望在这一章或这一本书中，告诉读者有关上面所述任务的全部内容，许多情况下，只能使用高级的黑箱程序包而别无选择。好的程序包是通用的，尽管他们并非都是用C写的。LINPACK 是由 Argonne 国家实验室开发的，并引起了特别的关注，因为它是公开发行的，有文档资料且免费使用。新版 LINPACK 现在也可得到了商品化的通用软件包(不必都用 C 语言写)，在 IMSL 和 NAG 图书馆中可以找到一些。

应当记住，高级的程序包都是为非常大型的线性系统求解而设计的。因而，为了减少运算次数和需要的存储空间，设计者们花费很大精力。按照输入系数矩阵的几种可能的简化形式：对称的，三角的，带状的，正定的等等，为各种任务提供了不同版本的例程。如果读者有一个大矩阵是这几种形式之一，则自然应当利用提供的这些不同的例程来提高效率，而不必使用为一般矩阵设计的形式。

有一个明显的界限，能把例程用的是**直接法**(即执行过程运算次数可判定)还是**迭代法**(即试图收敛到预定的解，但需很多步骤)区别出来。不论由于 $N$ 很大，还是由于问题是接近奇异的，当有效数字有丢失的危险时，迭代法更可取，在本书第2.7节和第18、19章中，我们讨论关于迭代法的内容。这些方法是重要的，但超出了我们的范围。然而，我们将讨论一种介于直接法和迭代法之间的技术，即解的迭代改进，它可以由直接法得到(第2.5节)。

## 2.1 高斯-约当消去法

求矩阵的逆时，**高斯-约当**(Gauss-Jordan)**消去法**与其它方法差不多一样有效。解线性方程组时，高斯-约当消去法既可产生具有一个或多个右端向量 **b** 的方程组的解，也可产生矩阵的逆 $\mathbf{A}^{-1}$。然而它的主要弱点在于：(i)需要同时存储和处理所有的右端项；(ii)不需要逆矩阵时，高斯-约当法比求解单一线性问题的最好技术(第2.3节)要慢三倍。这种方法的长处在于它跟其它直接法一样稳定，在使用完全主元法时可能更稳定些(参看下面内容)。

如果读者后面还要处理其它的右端向量，自然求矩阵逆较好，这样可以将右端向量乘以矩阵的逆。这确实给出了一个解答，但却很值得怀疑是否有舍入误差错误，它不如一开始就用右端向量求得的新向量结果那么好。

由于这些原因，在求解线性方程组或求矩阵的逆时，通常高斯-约当消去法不应成为首选方法。第2.3节中的分解法较佳。为什么我们还要给出高斯-约当法呢？这是因为它是直接了当的、易懂的、非常稳定的方法；而且在会出错而你又认为其可能为线性方程的解的时候，这种方法是一个极其好的"心理上"的后备。

一些人认为这种后备不仅仅是心理上的，高斯-约当法是一个"独立的"数值方法。这种说法在多数情况下被证明只是一种空想。除了下面所述的，在主元法选主元时相对稍有些差别外，高斯-约当消去法的实际操作过程与后两节中的例程的操作过程是密切相关的。

为清晰起见，并避免总写省略号(…)，我们下面写的方程组仅有四个方程和四个未知数，以及事先已知的三个不同的右端向量。用完全类似的形式，就可以写出更大一些的矩阵，并把方程组扩充成 $N\times N$ 矩阵，包含 $M$ 个右端向量。如下实现的例程自然是一般性的。

### 2.1.1 列增广矩阵消去法

考虑线性矩阵方程

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \cdot \left[ \begin{pmatrix} x_{11} \\ x_{21} \\ x_{31} \\ x_{41} \end{pmatrix} \sqcup \begin{pmatrix} x_{12} \\ x_{22} \\ x_{32} \\ x_{42} \end{pmatrix} \sqcup \begin{pmatrix} x_{13} \\ x_{23} \\ x_{33} \\ x_{43} \end{pmatrix} \sqcup \begin{pmatrix} y_{11} & y_{12} & y_{13} & y_{14} \\ y_{21} & y_{22} & y_{23} & y_{24} \\ y_{31} & y_{32} & y_{33} & y_{34} \\ y_{41} & y_{42} & y_{43} & y_{44} \end{pmatrix} \right]$$
$$= \left[ \begin{pmatrix} b_{11} \\ b_{21} \\ b_{31} \\ b_{41} \end{pmatrix} \sqcup \begin{pmatrix} b_{12} \\ b_{22} \\ b_{32} \\ b_{42} \end{pmatrix} \sqcup \begin{pmatrix} b_{13} \\ b_{23} \\ b_{33} \\ b_{43} \end{pmatrix} \sqcup \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \right] \tag{2.1.1}$$

这里圆点(·)表示矩阵相乘，而操作符⊔仅代表列增广，也就是说，去除相邻的小括号后，操作符⊔的操作数构成增广矩阵。

读者不应花很多时间来写方程(2.1.1)，而要明白它仅仅说明 $x_{ij}$ 是系数为 $b_{ij}$，$i=1,2,3,4$，第 $j$ 个右端项($j=1,2,3$)的向量解的第 $i$ 个元素($i=1,2,3,4$)；而未知系数 $y_{ij}$ 的矩阵是 $a_{ij}$ 矩阵的逆。换句话说，矩阵

$$[\mathbf{A}]\cdot[\mathbf{x}_1 \sqcup \mathbf{x}_2 \sqcup \mathbf{x}_3 \sqcup \mathbf{Y}] = [\mathbf{b}_1 \sqcup \mathbf{b}_2 \sqcup \mathbf{b}_3 \sqcup \mathbf{1}] \tag{2.1.2}$$

式中 **A** 和 **Y** 均为方阵，$\mathbf{b}_i$ 和 $\mathbf{x}_i$ 为列向量，**1** 为单位阵，它同时解决了线性方程组集合

$$\mathbf{A} \cdot \mathbf{x}_1 = \mathbf{b}_1 \quad \mathbf{A} \cdot \mathbf{x}_2 = \mathbf{b}_2 \quad \mathbf{A} \cdot \mathbf{x}_3 = \mathbf{b}_3 \tag{2.1.3}$$

以及

$$\mathbf{A} \cdot \mathbf{Y} = \mathbf{1} \tag{2.1.4}$$

现在来证实下列关于(2.1.1)的基本事实：

· 交换 **A** 的任意两行以及 **b** 和**1**的相应的行，并不改变(或任何形式的打乱)**x** 及 **Y** 的解。它不过是把同一线性方程组中，方程的次序调换而已。

· 同理，如果我们用 **A** 任意一行和其它行的线性组合代替该行，只要对 **b** 和**1**的相应行做相同的线性组合(当然**1**变换后不再是单位阵)，解集也不改变或受任何干扰。

· 交换 **A** 的任意两列，如果我们同时交换 **x** 和 **Y** 的相应的行，其解不变。换言之，这种交换搞乱了解的行的次序。如果是这样，我们需要把解的行恢复到原来次序以修正解。

高斯-约当消去法使用一个或更多的上述操作，把矩阵 **A** 归并成为单位矩阵。在化简完成后，从(2.1.2)中立即可以看出右端项即为解集。

## 2.1.2 选主元法

在"不选主元的高斯-约当消去法"中，仅使用了上面所列的第二项操作，它将第一行元素被元素 $a_{11}$除(可视为第一行与其他任意行的线性组合，只是对其他行用的是零系数)，然后用第一行乘以合适的系数去减其他各行，使得所有余下的 $a_{i1}$为零。至此，**A** 的第一列为单位阵的形式了。现在处理第二列，将第二行除以 $a_{22}$，然后将第一、三、四行分别减去其合适的倍数，使得它们第二列的元素为零。目前第二列也化简为单位阵的形式。类似地处理第三列和第四列。在对 **A** 做这些操作时，当然也要对 **b** 和**1**做相应的操作(变换后的**1**不会再有一点单位阵的样子了)。

显然，在使用对角线上的元素作除数时，如果该对角线上的元素为零，我们会遇到麻烦。(附带提一下，我们用作除数的元素被称为**主元素**或**主元**)。即便没有遇到零主元，事实上由于舍入误差的出现不选主元的高斯-约当消去法(没有用上述第一步和第三步过程)在数值上是不稳定。尽管这一点不是一目了然的，但确实如此，请读者务必不用不选主元的高斯-约当消去法(或是高斯消去法，见下面)！

那么，这种奇妙的选主元法是怎么回事呢？其实不过是进行行交换(**部分主元法**)或是行列均交换(**完全主元法**)，把一个特别需要的元素放到对角线的位置上来选取主元。因为我们不想搞乱已构造好的单位矩阵部分，所以要选择的元素必须满足：(i)在要标准化的行或其下面的行中，(ii)在要消去的列或其右边的列中。部分主元法比完全主元法要简单些，因为我们不必记下解向量的排列顺序。部分主元法仅当主元已处于正确的列时，是易于实现的。可以证明，在数学精度的意义上，部分主元法差不多与完全主元法效果一样好(进一步的论述和参考书目，见 Wilkinson 1965年的著作)。为给出二者的区别，我们在本节的例程中用完全主元法，在第2.3节中用部分主元法。

我们必须说明，如何识别一个元素是否是特别需要的主元素。这个问题不是理论上能够完全解答的。在理论上和实践中，大家知道简单地选取最大元素(指绝对值)是个很好的方法。然而这么做的费解之处是，主元的选取要依赖于方程组最初的形式。如果我们将方程组的第三个方程两边各乘以一百万，差不多可以保证第一个主元素在其中，而方程组的实际

解并不因做过乘法而改变。因而人们的习惯做法是，将所有方程左右两边进行缩放变换，使它们各自最大的系数标准化为单位值，然后再选作主元的元素也许已是最大了。这被称为**隐式主元法**。另外需有记录保存每行所乘的比例因子。(第2.3节的例程中包含了隐式主元法，但本节的例程中没有)。

最后，让我们来考虑一下这种方法的存储要求。略加思考，可以知道，在算法的每一步中，要么 **A** 的元素是1或是0(如果它已经是化简成的单位矩阵的一部分)，要不然由单位矩阵**1**所开始的矩阵中，相应元素会变成1或是0(如果 **A** 中的对应元素还未被化简为单位矩阵形式)。因此，矩阵**1**不必单独存储；矩阵 **A** 的逆在 **A** 中逐步构造，同时破坏 **A** 原来的内容。同理，解向量 **x** 能够逐步替代右端项向量 **b**，且使用同一存储空间，因为每一列 **A** 中的元素归并后，**b** 中相应的行项便不再使用。

下面的例程是用完全主元素法的高斯-约当消去法：

```
#include  <math.h>
#include  "nrutil.h"
#define SWAP(a,b){ temp=(a);(a)==(b);(b)==temp;}

Void gaussj (float **a,int n,float **b,int m)
    对上面(2.1.1)方程用高斯-约当消去法求解线性方程组。a[1..n][1..n]为一n×n输入矩阵。b[1..n][1..m]为包
    含m个右端项向量的n×m输入矩阵。输出时，a被其逆矩阵代替，b被相应的解向量所代替。
{
    int *indxc, *indxr, *ipiv;
    int i, icol, irow, j, k, l, ll;
    float big, dum, pum, pivinv temp;

    indxc=ivector(1,n);                          整形数组ipiv,indxr和indxc用于选主元作记录
    indxr=ivector(1,n);
    ipiv=ivector(1,n);
    for (j=1;i<=n;j++) ipiv[j]=0;
    for (i=1;i<=n;i++) {                         归并列的主循环
        big=0.0;
        for (j=1;j<=n;j++)                       找主元素的外层循环
            if (ipiv[j] != 1)
                for (k=1;k<=n;k++){
                    if (ipiv[k] == 0) {
                        if (fabs(a[j][k] >=big) {
                            big = fabs(a[j][k]);
                            irow=j;
                            icol=k;
                        }
                    } else if (ipiv[k] > 1) nrerror("gaussj;Singuler Martix-1");
                }
        ++(ipiv[icol]);                          至此我们已求得主元素，如果需要的话，进行换行把主元素放到对角
                                                 线的位置上。列并不做实际交换，只是进行重新标注：indxc[i]为第i
                                                 个主元素所在的列，即被化简的第i列；而indxr[i]是主元素原来位
                                                 于的行。如果indxr[i]≠indxc[i]，意味着要做列交换。用这种形式记
                                                 录，解集b最终的次序是正确的，而逆矩阵的列次序被打乱
        if (ifow != icol){
            for (l=1;l<=n;l++) SWAP(a[irow][l],a[icol][l])
            for (l=1;l<=m;l++) SWAP(b[irow][l],b[icol][l])
        }
        indxr[i]=irow;                           用位于irow行icol列的主元素去除其所在的行
        indxc[i]=icol;
        if (a[icol][icol] == 0.0) nrerror("gaussj: Singular Martix-2");
```

```
        pivinv=1.0/a[icol][icol];
        a[icol][icol]=1.0;
        for (l=1;l<=n;l++) a[icol][l] *= pivinv;
        for (l=1;l<=m;l++) b[icol][l] *= pivinv;
        for (ll=1;ll<=n;ll++)                  下面进行行约简
            if (ll != icol) {                  当然主元素除外
                dum=a[ll][icol];
                a[ll][icol]=0.0;
                for (l=1;l<=n;l++) a[ll][l] -= a[icol][l]*dum;
                for (l=1;l<=m;l++) a[ll][l] -= b[icol][l]*dum;
            }
    }                                          列归并主循环的结尾。考虑已进行过列交换，为使解向量保持原
    for (l=n;l>=1;l--) {                       来的次序，再根据其交换的相反顺序交换各列，以整理恢复解
        if (indxr[l] != indxc[l])
            for (k=1;k<=n;k++)
                SWAP(a[k][indxr[l]],a[k][indxc[l]]);
    }                                          程序结束
    free ivector(ipiv,1,n);
    free ivector(indxr,1,n);
    free inector(indxc,1,n);
}
```

### 2.1.3 行和列消去法策略

以上的讨论可以用下面形式加以发挥，用某简单矩阵 **R** 前乘（即左乘）以矩阵 **A** 来完成行操作。例如，矩阵 **R** 具有分量

$$R_{ij}=\begin{cases}1 & \text{当 } i=j \text{ 和 } i\neq 2,4\\ 1 & \text{当 } i=2,j=4\\ 1 & \text{当 } i=4,j=2\\ 0 & \text{其它}\end{cases}\tag{2.1.5}$$

其效果是第2行和第4行交换。单独用行操作的高斯-约当消去法（包括部分主元法的可能性）可由一系列左乘组成：

$$\begin{aligned}\mathbf{A}\cdot\mathbf{x}&=\mathbf{b}\\(\cdots\mathbf{R}_3\cdot\mathbf{R}_2\cdot\mathbf{R}_1\cdot\mathbf{A})\cdot\mathbf{x}&=\cdots\mathbf{R}_3\cdot\mathbf{R}_2\cdot\mathbf{R}_1\cdot\mathbf{b}\\(\mathbf{1})\cdot\mathbf{x}&=\cdots\mathbf{R}_3\cdot\mathbf{R}_2\cdot\mathbf{R}_1\cdot\mathbf{b}\\\mathbf{x}&=\cdots\mathbf{R}_3\cdot\mathbf{R}_2\cdot\mathbf{R}_1\cdot\mathbf{b}\end{aligned}\tag{2.1.6}$$

关键在于因为这些矩阵 **R** 从右向左构造，所以在每一步中，等式右边是从一个向量简单地换成另一向量。

同样，列操作可以相应地用简单的矩阵称为 C 后乘（或右乘）矩阵 **A** 实现，若式(2.1.5)中的矩阵右乘以矩阵 **A**，则使矩阵 **A** 中第二列与第四列交换。列操作的消去法是，（概念上）在矩阵 **A** 和未知向量 **x** 之间插入列操作项及其逆操作。

$$\begin{aligned}\mathbf{A}\cdot\mathbf{x}&=\mathbf{b}\\\mathbf{A}\cdot\mathbf{C}_1\cdot\mathbf{C}_1^{-1}\cdot\mathbf{x}&=\mathbf{b}\\\mathbf{A}\cdot\mathbf{C}_1\cdot\mathbf{C}_2\cdot\mathbf{C}_2^{-1}\cdot\mathbf{C}_1^{-1}\cdot\mathbf{x}&=\mathbf{b}\\(\mathbf{A}\cdot\mathbf{C}_1\cdot\mathbf{C}_2\cdot\mathbf{C}_3\cdots)\cdots\mathbf{C}_3^{-1}\cdot\mathbf{C}_2^{-1}\cdot\mathbf{C}_1^{-1}\cdot\mathbf{x}&=\mathbf{b}\\(\mathbf{1})\cdots\mathbf{C}_3^{-1}\cdot\mathbf{C}_2^{-1}\cdot\mathbf{C}_1^{-1}\cdot\mathbf{x}&=\mathbf{b}\end{aligned}\tag{2.1.7}$$

因而得解为：

$$\mathbf{x}=\mathbf{C}_1\cdot\mathbf{C}_2\cdot\mathbf{C}_3\cdots\mathbf{b}\tag{2.1.8}$$

要注意(2.1.8)和(2.1.6)式之间的本质区别。在(2.1.8)式中，这一系列矩阵 **C** 是以倒次序作用于向量 **b**，从而得到解向量。也就是它们必须依照这种倒次序存储。这一需求降低了列操作的实用性，例如在完

全主元法的情况下,也限制了它们的使用。

参考文献和进一步读物

Wilkinson, J. H. 1965, *The Algebraic Eigenvalue Problem* (New York: Oxford University Press). [1]

## 2.2 具有回代过程的高斯消去法

有回代过程的高斯消去法的用途主要是适合于教学。它介于象高斯-约当这样的完全消去法和下章讨论的三角分解法之间。高斯消去法并不是把矩阵完全化简为单位阵,而是半单位阵,即其对角线上和(比方说)其上半部的内容不为空。现在我们来看一下这么做的好处。

假定在做高斯-约若当消去时,如第2.1节所述,我们每一步仅减去位于当前主元素下面的那些行。比如,当 $a_{22}$ 为主元素时,我们用它的值去除第二行(跟以前一样),但现在用主元素所在行仅把 $a_{32}$ 和 $a_{42}$ 消为零,而不消 $a_{12}$(参见方程2.1.1)。再假定我们仅作部分主元法消去,不进行列交换。这样未知量的顺序就不必修改。

对所有主元素处理完后,得到化简后的方程如下(对单个右端向量的情况):

$$\begin{bmatrix} a'_{11} & a'_{12} & a'_{13} & a'_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & 0 & a'_{33} & a'_{34} \\ 0 & 0 & 0 & a'_{44} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \end{bmatrix} \tag{2.2.1}$$

这里的撇号表示 $a$ 和 $b$ 不再是原来的数值,而是经过所有行消去操作后,得到的修改了的值。至此,高斯消去过程结束。

### 2.2.1 回代过程

但我们如何求解 $x$ 呢?最后一个 $x$(例中为 $x_4$)已可独立求出,即

$$x_4 = b'_4/a'_{44} \tag{2.2.2}$$

最后一个 $x$ 知道后便可来求倒数第二个 $x$,

$$x_3 = \frac{1}{a'_{33}}[b'_3 - x_4 a'_{34}] \tag{2.2.3}$$

然后,继续求其前面的 $x$。典型步骤为

$$x_i = \frac{1}{a'_{ii}}\left[b'_i - \sum_{j=i+1}^{N} a'_{ij} x_j\right] \tag{2.2.4}$$

方程式(2.2.4)定义的过程称为**回代过程**。高斯消去法与回代过程结合起来即可求得方程组的解。

与高斯-约当消去法相比,高斯消去法与回代过程相结合的优点在于它的运算次数少,要快一些。高斯-约当消去法的最内层循环每次包含一次减法和一次乘法,需要执行 $N^3$ 次和 $N^2M$ 次(当有 $N$ 个方程式和 $M$ 个未知数时)。而高斯消去法的相应循环仅执行 $\frac{1}{3}N^3$ 次及 $\frac{1}{2}N^2M$ 次(仅化简矩阵的一半,且要变成零的元素个数减少到三分之一)。每个右端项可代要

执行$\frac{1}{2}N^2$次类似的循环（一次乘法加一次减法）。对 $M<<N$ 的高斯消去法（即仅有几个右端项的情况）差不多要比高斯-约当法好三倍，（我们把这个好处打点折扣减到1.5倍，因为它没有象高斯-约当法那样能计算出逆矩阵）。

要计算矩阵的逆时（我们可视其为 $M=N$ 个右端项的情况，即单位阵的列看作 $N$ 个单位向量），一眼可以看出，高斯消去法和回代法需要$\frac{1}{3}N^3$（矩阵化简）$+\frac{1}{2}N^3$（右端项处理）$+\frac{1}{2}N^3$（$N$ 次回代）$=\frac{4}{3}N^3$次循环操作，这比高斯-约当法的 $N^3$次要多。然而，单位向量是很特殊的，除了一个元素是1外，其余全是0元素。如果将此考虑在内，右端项处理可减为仅$\frac{1}{6}N^3$次循环操作，故对求矩阵的逆来说，这两种方法效率是同样。

高斯消去法和高斯-约当消去法都有一个缺点，即右端项必须提前知道。下一节中要讨论的 $LU$ 分解法就没有这个缺点，不管是对有任意个右端项的求解问题，还是求矩阵的逆，这种方法的运算次数都要少。由于这个原因，我们就不写出高斯消去法的实现程序了。

## 2.3 *LU* 分解和它的应用

假定我们能够把矩阵 **A** 写成两个矩阵相乘的形式

$$\mathbf{L}\cdot\mathbf{U}=\mathbf{A} \tag{2.3.1}$$

其中 **L** 为**下三角矩阵**（仅在对角线上及其下面有元素），**U** 为**上三角矩阵**（仅在对角线上及其上面有元素）。举个例子，对4×4矩阵的 **A** 的情况，方程(2.3.1)如下：

$$\begin{bmatrix}\alpha_{11} & 0 & 0 & 0\\ \alpha_{21} & \alpha_{22} & 0 & 0\\ \alpha_{31} & \alpha_{32} & \alpha_{33} & 0\\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \alpha_{44}\end{bmatrix}\cdot\begin{bmatrix}\beta_{11} & \beta_{12} & \beta_{13} & \beta_{14}\\ 0 & \beta_{22} & \beta_{23} & \beta_{24}\\ 0 & 0 & \beta_{33} & \beta_{34}\\ 0 & 0 & 0 & \beta_{44}\end{bmatrix}=\begin{bmatrix}a_{11} & a_{12} & a_{13} & a_{14}\\ a_{21} & a_{22} & a_{23} & a_{24}\\ a_{31} & a_{32} & a_{33} & a_{34}\\ a_{41} & a_{42} & a_{43} & a_{44}\end{bmatrix} \tag{2.3.2}$$

我们可以用象式(2.3.1)的分解来解线性方程组

$$\mathbf{A}\cdot\mathbf{x}=(\mathbf{L}\cdot\mathbf{U})\cdot\mathbf{x}=\mathbf{L}\cdot(\mathbf{U}\cdot\mathbf{x})=\mathbf{b} \tag{2.3.3}$$

首先求解向量 **y** 使得

$$\mathbf{L}\cdot\mathbf{y}=\mathbf{b} \tag{2.3.4}$$

然后再来求解

$$\mathbf{U}\cdot\mathbf{x}=\mathbf{y} \tag{2.3.5}$$

把一个线性方程组拆成两个连续的方程组有什么好处呢？好处在于求解一个三角形的方程组相当容易，正如我们在第2.2节中（方程(2.2.4)已经见到的那样）。而方程(2.3.4)可用向前替换过程求解如下：

$$\begin{aligned}y_1&=\frac{b_1}{\alpha_{11}}\\ y_i&=\frac{1}{\alpha_{ii}}\left[b_i-\sum_{j=1}^{i-1}\alpha_{ij}y_j\right]\qquad i=2,3,\cdots,N\end{aligned} \tag{2.3.6}$$

方程(2.3.5)可用回代过程求解,跟方程(2.2.2)~方程(2.2.4)一样,

$$x_N = \frac{y_N}{\beta_{NN}}$$

$$x_i = \frac{1}{\beta_{ii}}\left[y_i - \sum_{j=i+1}^{N} \beta_{ij} x_j\right] \qquad i = N-1, N-2, \cdots, 1 \tag{2.3.7}$$

方程组(2.3.6)和(2.3.7)共需执行 $N^2$次内层循环(对每个右端项 **b**),每个内层循环包括一次乘法和一次加法。如果我们有 $N$ 个右端项,它们是单位列向量(在求矩阵逆时就是这种情况),考虑这些零元素可把方程(2.3.6)的总的执行次数从$\frac{1}{2}N^3$减少到$\frac{1}{6}N^3$,而方程(2.3.7)的执行次数不变,仍为$\frac{1}{2}N^3$。

注意,一旦对 **A** 进行了 $LU$ 分解,我们就可以一次求解所有要解的右端项,跟第2.1节和第2.2节中的方法比,这是一个显著的优点。

### 2.3.1 进行 *LU* 分解

对给定的 **A** 我们怎样才能求得 **L** 和 **U** 呢?首先,我们写出方程(2.3.1)或(2.3.2)的第 $i,j$ 项分量。它总是一个和的形式,开始部分形式如下

$$\alpha_{i1}\beta_{1j} + \cdots = a_{ij}$$

和式中的项数依赖于 $i$ 和 $j$ 中较小的数。事实上有三种形式,

$$i < j: \qquad \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \cdots + \alpha_{ii}\beta_{ij} = a_{ij} \tag{2.3.8}$$

$$i = j: \qquad \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \cdots + \alpha_{ii}\beta_{jj} = a_{ij} \tag{2.3.9}$$

$$i > j: \qquad \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \cdots + \alpha_{ij}\beta_{jj} = a_{ij} \tag{2.3.10}$$

方程(2.3.8)~(2.3.10)共有 $N^2$个方程式,而要求 $N^2+N$ 个未知的 $\alpha$ 和 $\beta$(因对角线的未知元有两套)。既然未知数的个数比方程个数多,我们就人为地指定 $N$ 个未知数,然后再来求解其它的未知数。事实上,我们总是令

$$\alpha_{ii} = 1 \qquad i = 1,2,\cdots,N \tag{2.3.11}$$

这里有一个奇妙的步骤叫做**克鲁特算法**(Crout's algorihm),它仅通过按某种次序排列方程,就能相当容易地求出方程(2.3.8)~(2.3.11)的 $N^2+N$ 个方程中的所有 $\alpha$ 和 $\beta$!这个步骤如下:

- 设 $\alpha_{ii}=1, i=1\cdots,N$(方程2.3.11)
- 对每个 $j=1,2,3,\cdots,N$ 做以下两步:首先对每 $i=1,2,\cdots,j$,用方程(2.3.8),(2.3.9)和(2.3.11)来解 $\beta_{ij}$,即

$$\beta_{ij} = a_{ij} - \sum_{k=1}^{i-1} \alpha_{ik}\beta_{kj} \tag{2.3.12}$$

(或方程(2.3.12)中 $i=1$时,求和项为零),第二步,对每个 $i=j+1,j+2,\cdots,N$ 用方程(2.3.10)来求解 $\alpha_{ij}$,即

$$\alpha_{ij} = \frac{1}{\beta_{jj}}\left(a_{ij} - \sum_{k=1}^{j-1} \alpha_{ik}\beta_{kj}\right) \tag{2.3.13}$$

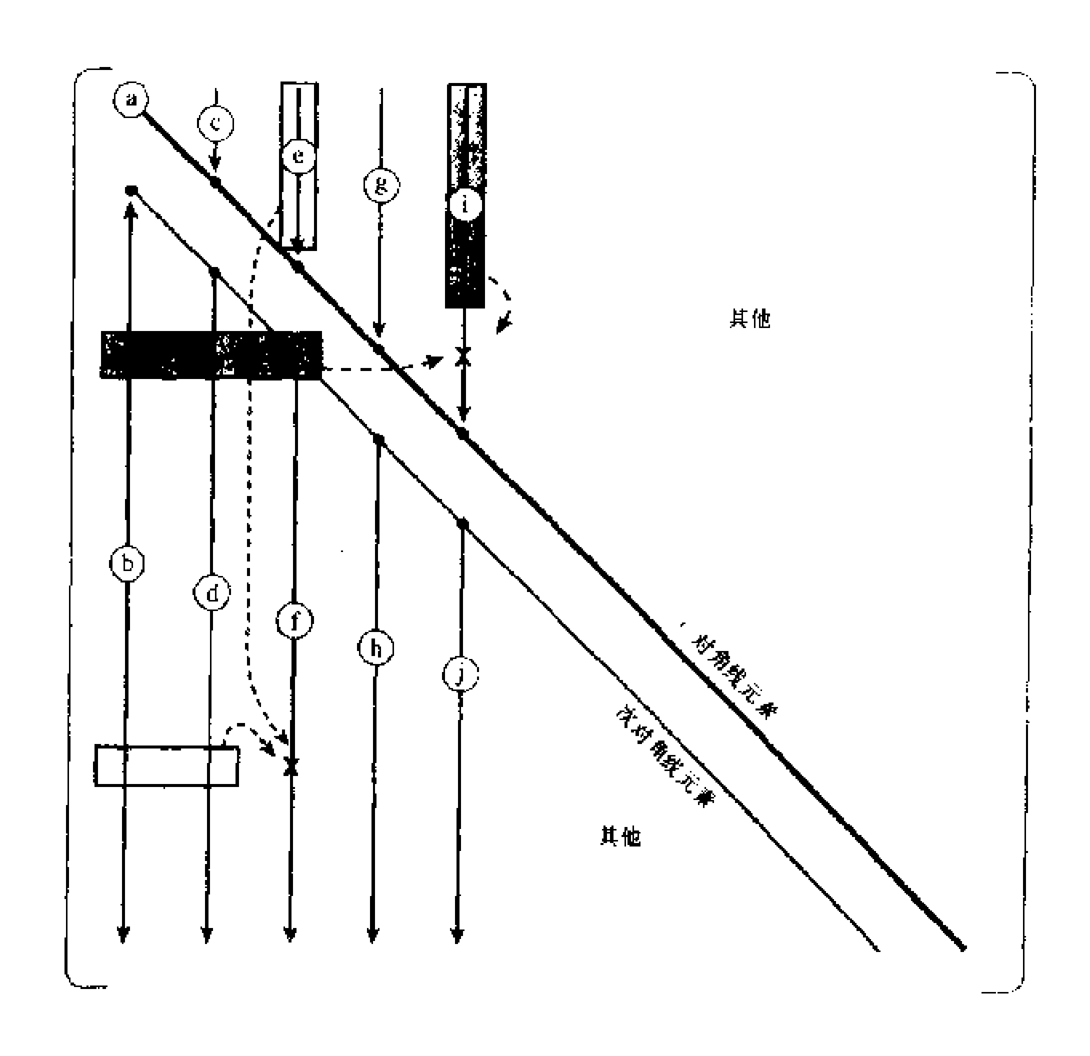
在进行下一个j之前要保证做过以上两步。

如果按上述过程做过几次迭代后,就会发现,方程(2.3.12)和(2.3.13)右端的 $\alpha$ 与 $\beta$ 在

需要时已经知道了。还会看到，每一个 $a_{ij}$ 仅被使用一次便不再用。这意味着相应的 $\alpha_{ij}$ 和 $\beta_{ij}$ 可以存储在 $a$ 所占的存储空间：即分解是"同址"进行的。主对角线上的单位元素 $\alpha_{ii}$（方程2.3.11）并不存储。简而言之，克鲁特算法得到的是 $\alpha$ 和 $\beta$ 的混合矩阵，其排列是

$$\begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ \alpha_{21} & \beta_{22} & \beta_{23} & \beta_{24} \\ \alpha_{31} & \alpha_{32} & \beta_{33} & \beta_{34} \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \beta_{44} \end{bmatrix} \tag{2.3.14}$$

列从左向右，每一列从上到下（参看图2.3.1）。




原始矩阵的元素按小写字母的顺序被修改：a、b、c、等等。带阴影的矩形方块表示那些已被修改的元素，它们被用来修改标有"×"记号的两个典型元素

图2.3.1 克鲁特算法对矩阵进行 $LU$ 分解

至于主元法（即选择一个合适的主元素去除方程(2.3.13)的两边），它对克鲁特方法的稳定性是绝对必要的。只有部分主元法（行交换）可以有效地实现。然而，这已足够使算法稳定了。顺便提一下，这意味着我们并不是把矩阵 **A** 分解成为 $LU$ 的形式，而是将其按行置换

的方式分解(如果我们保存这个置换的次序,该分解就与原来的分解一样有用)。

克鲁特算法中的主元法有点精妙之处。关键是要注意方程(2.3.12),在 $i=j$(最后一个方程)时,与方程(2.3.13)(除了后者还要做一次除法外)是完全一样的,这两种情况求和的上限都是 $k=j-1(=i-1)$。这意味着,我们不必费心去考虑对角线元素 $\beta_{jj}$ 是否会正好落在对角线上,也不必考虑该列中,它下面的某个元素(未做除法的)$\alpha_{ij}$,$i=j+1,\cdots,N$ 是否会"提升"成为对角元素 $\beta$。当所有该列元素都求得后,这点便会证实。现在应该能够猜到,我们将选取最大的元素作为对角线元素 $\beta$(主元素),然后用它去除其它元素。这便是**部分主元法的克鲁特方法**。我们的实现方法中还有一个精妙之处:它首先找每行的最大元素,而后(在找最大主元素时)乘以一个比例系数,就象我们最初对所有方程进行比例变换,使它们最大系数成为单位元素那样。这就是第2.1节中提到过的**隐式主元法**。

```
#include  <math.h>
#include  "nrutil.h"
#define   TINY   1.0e-20;              一个小数

void  ludcmp(float **a,int n,int *indx,float *d)
```

对给定的一个 n×n 矩阵 a[1..n][1..n],本程序对其进行按行置换的 LU 分解,结果存在该矩阵中。a 和 n 是输入量,a 为输出结果,其元素的排列形式与上面的(2.3.14)一样。indx[1..n]为输出向量,它用来记录因部分主元法而改变了的行排列次序。输出变量 d 的值为 ±1,相应地表示行交换次数为偶数还是奇数。本程序跟程序 **lubksb** 一起用来求解线性方程组或者求矩阵的逆。

```
{
    int i,imax,j,k;
    float big,dum,sum,temp;
    float *vv;                                      vv 用于保存每行的内含比例因子

    vv=vector(1,n);
    *d=1.0;                                         还未进行交换
    for (i=1;i<=n;i++) {                            按行循环,求内含的比例因子
        big=0.0;
        for(j=1;j<=n;j++)
            if((temp=fabs(a[i][j])) > big) big=temp;
        if (big == 0.0) nrerror("Singular matrix in routine ludcmp");    没有非零的最大元素
        vv[i]=1.0/big;                              保存比例因子
    }
    for (j=1;j<=n;j++){                             克鲁特方法的列循环
        for (i=1;i<j;i++) {                         方程2.3.12中的 i≠j 的情况
            sum=a[i][j];
            for (k=1;k<i;k++) sum -= a[i][k]*a[k][j];
            a[i][j]=sum;
        }
        big=0.0;                                    初始化最大主元素变量
        for (i=j;i<=n;i++) {                        方程2.3.12 i=j 的情况以及方程
            sum=a[i][j];                            2.3.13 i=j+1,...,N 的情况
            for (k=1;k<j;k++)
                sum -= a[i][k]*a[k][j];
            a[i][j]=sum;
            if ( (dum=vv[i]*fabs(sum)) >= big) {    新的候选主元是否比已选中的更合适
                big=dum;
                imax=i;
            }
        }
        if (j != imax) {                            需要进行交换吗
            for (k=1;k<=n;k++) {                    是的,进行交换
                dum=a[imax][k];
```

```
                a[imax][k]=a[j][k];
                a[j][k]=dum;
            }
            *d = -(*d);                                    改变 d 的奇偶性
            vv[imax]=vv[j];                                还要交换比例因子
        }
        indx[j]=imax;
        if (a[j][j] == 0.0) a[j][j]=TINY;                  最后除以主元素
        if (j != n) {                                      如果主元素为零则矩阵是奇异的(至
            dum=1.0/(a[j][j]);                             少对本算法精度而言),对奇异矩阵
            for (i=j+1;i<=n;i++) a[i][j] *= dum;           的情况需要把 TINY 换成零.
        }
    }                                                      转回归并下一列
    free_vector(vv,1,n);
}
```

下面的程序用向前替换过程和回代过程来实施方程(2.3.6)和(2.3.7)。

```
void lubksb(float **a,int n,int *indx,float b[])
```

求解 $n$ 维线性方程 **A**·**x**=**B**。其中 a[1..n][1..n]为输入量,它不是原始矩阵 **A**,而是经程序 **ludcmp** 后生成的其 $LU$ 分解阵,indx[1..n]为输入量,是 **ludcmp** 返回的记录行排列顺序的向量 b[1..n]为输入右端项向量 **B**,返回解向量 **x**。a,n 和 indx 不被该程序修改,可留下来再被调用,求不同的右端项 b 的解。这个例程考虑了 b 可能开始有很多零元素的情况,因而在求矩阵的逆时可提高效率。

```
{
    int i,ii=0,ip,j;
    float sum;

    for (i=1;i<=n;i++) {                  当 ii 为正数时,它表示 b 的第一个非零元的位置。
        ip=indx[i];                       现在来做方程(2.3.6)的向前替换。其唯一的缺点是没有
        sum=b[ip];                        恢复原来的排列顺序。
        b[ip]=b[i];
        if (ii)
            for (j=ii;j<=i-1;j++) sum -= a[i][j]*b[j];
        else if (sum) ii=i;               遇到非零元素,下面要进入上面的求和循环
        b[i]=sum;
    }
    for (i=n;i>=1;i--) {                  现在来做方程(2.3.7)的回代
        sum=b[i];
        for (j=i+1;j<=n;j++) sum -= a[i][j]*b[j];
        b[i]=sum/a[i][i];                 保存解向量 x 的一个分量
    }
}
```

程序 **ludcmp** 中,$LU$ 分解大约需要执行 $\frac{1}{3}N^3$ 次内层循环(每次包括一次乘法和一次加法)。这是求解一个(或少量几个)右端项时的运算次数,它要比第2.1节中的高斯—若当程序 **gaussj** 要快三倍,比不计算逆矩阵的高斯—若当程序快一点五倍。当要求逆矩阵时,总的运算次数(包括上面方程(2.3.7)后面讨论的向前替换和回代部分)为$(\frac{1}{3}+\frac{1}{6}+\frac{1}{2})N^3=N^3$,与 **gaussj** 相同。

总结一下,求解线性方程组 **A**·**x**=**b** 的可取的方法是:

```
float **a,*b,d;
int n,*indx;
...
ludcmp(a,n,indx,&d);
```

```
lubksb(a,n,indx,b);
```

解 **x** 将保存在 b 中，且原来的矩阵 **A** 会被破坏。

如果随后还想求解具有相同的 **A** 但右端项 **b** 不同的线性方程组，则可以仅仅重写

```
lubksb(a,n,indx,b);
```

当然矩阵 **A** 已非原来的形式，这里的 a 和 indx 已经由程序 **ludcmp** 设置好了。

### 2.3.2 矩阵的求逆

使用上一节中的 $LU$ 分解和回代过程的程序，能完全直接地按列来求矩阵的逆(没有更好办法来求了)。

```
#define N ...
float **a,**y,d,*col;
int i,j,*indx;
...
ludcmp(a,N,indx,&d);                         仅把矩阵分解一次
for(j=1;j<=N;j++) {                          按列求逆
    for(i=1;i<=N;i++) col[i]=0.0;
    col[j]=1.0;
    lubksb(a,N,indx,col);
    for(i=1;i<=N;i++) y[i][j]=col[i];
}
```

矩阵 y 保存的是原始矩阵 a 的逆，矩阵 a 会被破坏。使用象 **gaussj**(第2.1节)这样的高斯-约当例程求矩阵的逆也是不错的选择，它同样也破坏原始矩阵。这两种方法的运算次数差不多相同。

如果碰巧要根据矩阵 **A** 和 **B** 来计算 $\mathbf{A}^{-1}\cdot\mathbf{B}$，应当对 **A** 进行 $LU$ 分解，然后用 **B** 的列进行回代，而不应用单位向量先求 **A** 的逆。这样可以省去一次整个矩阵的乘法运算，并且精度更高。

### 2.3.3 矩阵的行列式

经 $LU$ 分解的矩阵的行列式仅为对角线上元素的积：

$$\det = \prod_{j=1}^{N} \beta_{jj} \tag{2.3.15}$$

回顾一下，我们并没有求原始矩阵的分解，而是只进行过行交换后的分解。幸运的是，我们记录了行交换次数是奇数还是偶数，因此，我们就可以在乘积前面冠以相应的符号。(现在终于知道第2.3节中的程序 ludcmp 设变量 d 的原因了)。

计算行列式须调用一次 **ludcmp**，而不必接着调用 **lubksb** 进行回代了。

```
#define N ...
float **a,d;
int j,*indx;
...
ludcmp(a,N,indx,&d);                  返回值 d 为±1
for(j=1;j<=N;j++) d *= a[j][j];
```

d 中现在保存的即为原始矩阵 **a** 的行列式值，**a** 将会被破坏。

对实际出现的大矩阵，很可能出现行列式值上溢或下溢，超出了计算机所能表示的浮点数的范围。在这种情况下，可以修改上述程序中的循环部分，(例如)除以十的幂并单独保存该比例因子，或(例如)求出各乘数绝对值的对数和并单独保存其符号。

### 2.3.4 复数系统方程

如果矩阵 **A** 是实数的，而右端向量是复数的，即 $\mathbf{b}+i\mathbf{d}$，则(i)用通常的方法对 **A** 进行 $LU$ 分解，(ii)用 **b** 回代过程求得解向量的实部，(iii)用 **d** 回代过程求得解向量的虚部。

如果矩阵本身是复数的，就是说要求解的方程为：

$$(\mathbf{A}-i\mathbf{C})\cdot(\mathbf{x}+i\mathbf{y})=(\mathbf{b}-i\mathbf{d}) \tag{2.3.16}$$

则有两种可能的办法求解。最好的办法是重写 **ludcmp** 和 **lubksb** 为复数程序，在构造比例向量 vv 及寻找最大主元素时，用复数的模代替绝对值。其它部分都很显然，需要时就进行复数的算术运算。(参见第1.2节及5.4节有关 C 语言写的复数算术运算的讨论。)

快速但不太好的求解复数系统的办法是，将方程(2.3.16)的实部和虚部分开，即

$$\begin{aligned}\mathbf{A}\cdot\mathbf{x}-\mathbf{C}\cdot\mathbf{y}&=\mathbf{b}\\ \mathbf{C}\cdot\mathbf{x}+\mathbf{A}\cdot\mathbf{y}&=\mathbf{d}\end{aligned} \tag{2.3.17}$$

它可以写成一个 $2N\times 2N$ 的实数方程，

$$\begin{pmatrix}\mathbf{A} & -\mathbf{C}\\ \mathbf{C} & \mathbf{A}\end{pmatrix}\cdot\begin{pmatrix}\mathbf{x}\\ \mathbf{y}\end{pmatrix}=\begin{pmatrix}\mathbf{b}\\ \mathbf{d}\end{pmatrix} \tag{2.3.18}$$

然后用 **ludcmp** 和 **lubksb** 来求解目前的形式。该方案在存储上要花两倍的空间，因为 **A** 和 **C** 都被存储了两次。另外，也要花两倍的时间，因为在用复数形式写成的程序中，复数乘法仅需4次实数相乘，而对 $2N\times 2N$ 的问题求解是 $N\times N$ 问题的工作量的8倍。如果对这两方面的代价能够容忍的话，则方程(2.3.18)也是一种很方便的求解办法。

## 2.4 三对角及带状对角系统方程

线性方程系统的特例之一是**三对角形式**的，也就是说，非零的元素仅出现在对角线及其上、下一列的位置上，这种情况会经常出现。**带状对角**形式的系统也常见，其非零的元素仅出现在主对角线附近(上或下)的几个对角行中。

对三对角系统，$LU$ 分解过程，前代或回代每次只需 $O(N)$ 次操作，整个求解过程可以非常简洁地编写。求解程序 **tridag** 在后面的章节中还要用到。

自然不必把整个 $N\times N$ 矩阵全部存起来，而只需保存非零的元素，存作三个向量。要求解的方程为：

$$\begin{bmatrix}b_1 & c_1 & 0 & \cdots & & & \\ a_2 & b_2 & c_2 & \cdots & & & \\ & & & \cdots & & & \\ & & & \cdots & a_{N-1} & b_{N-1} & c_{N-1}\\ & & & \cdots & 0 & a_N & b_N\end{bmatrix}\cdot\begin{bmatrix}u_1\\ u_2\\ \cdots\\ u_{N-1}\\ u_N\end{bmatrix}=\begin{bmatrix}r_1\\ r_2\\ \cdots\\ r_{N-1}\\ r_N\end{bmatrix} \tag{2.4.1}$$

注意 $a_1$ 和 $c_N$ 没有定义，下面的程序并不参考它们。

```
#include "nrutil.h"

void tridag(float a[], float b[], float c[], float r[], float u[], unsigned long n)
    求解向量u[1..n]，三对角线性矩阵由方程(2.4.1)给出。a[1..n]、b[1..n]、c[1..n]、及r[1..n]为输入向量，并不被
    修改。
{
    unsigned long j;
    float bet, *gam;

    gam=vector(1,n);                                        gam需要一个向量的工作空间
    if (b[1] == 0.0) nrerror("Error 1 in tridag");          若这种情况发生，应重写方程，u2忽略
    u[1]=r[1]/(bet=b[1]);                                   不计，成为N-1维的
    for (j=2;j<=n;j++) {                                    分解及前代
        gam[j]=c[j-1]/bet;
        bet=b[j]-a[j]*gam[j];
        if (bet == 0.0) nrerror("Error 2 in tridag");       算法失败，见下面的解释
        u[j]=(r[j]-a[j]*u[j-1])/bet;
    }
    for (j=(n-1);j>=1;j--)
        u[j] -= gam[j+1]*u[j+1];                            回代
    free_vector(gam,1,n);
}
```

在 **tridag** 中没有求主元。这是因为，如果对非奇异的矩阵，**tridag** 也失败的话，则对非奇异矩阵就会有可能遇到零主元的。在实践中，不必为此过于担心。那些能产生三对角线性系统的问题通常都有其它的属性，它能保证算法 **tridag** 会成功。举个例子，如果

$$|b_j| > |a_j| + |c_j| \qquad j = 1, \cdots, N \tag{2.4.2}$$

（称作**对角线优势**），则可以证明算法不会遇到零主元。

可以构造出特殊的例子，来说明由于算法没找主元而导致数值不稳定。然而，实践中这种不稳定几乎从未遇到过，不象一般的矩阵问题中，求主元是必不可少的。

三对角算法是算法中的一种稀有情况，实践中其稳健性要比理论上强得多。当然，一旦遇到了 **tridag** 算法失败了的问题，则可以换用更一般的方法，即求解带状对角系统的方法，下面就要讨论它（程序 **bandec** 和 **banbks**）。

一些其它的矩阵形式，包含了三对角及少数几个附加的非零元素（如右上角或左下角）也可以进行快速求解，参见第2.7节。

## 2.4.1 带状对角系统

三对角系统的非零元仅出现在对角线及上、下一个元素的位置上；而带状对角系统则稍更一般些，紧靠在对角线左边（下边）有（例如）$m_1 \geqslant 0$个非零元素，紧靠其右边（上边）有 $m_2 \geqslant 0$个非零元素。当然，这仅当 $m_1$和 $m_2$都<<N 时才有意义。在这种情况下，用 $LU$ 分解求解线性系统可以比通用 $N \times N$ 的情况完成得更快，占空间更少。

用元素 $a_{ij}$对带状对角矩阵的精确定义为：

$$a_{ij} = 0 \quad 当 \quad j > i + m_2 \quad 或 \quad i > j + m_1 \tag{2.4.3}$$

带状对角矩阵用一种称为压缩形式的方式进行存储及操作。把矩阵顺时针旋转45°，使非零元素排成一个细长的矩阵，它有 $m_1 + 1 + m_2$列、N 行。最好用个例子来解释一下：

带状对角矩阵

$$
\begin{pmatrix}
3 & 1 & 0 & 0 & 0 & 0 & 0 \\
4 & 1 & 5 & 0 & 0 & 0 & 0 \\
9 & 2 & 6 & 5 & 0 & 0 & 0 \\
0 & 3 & 5 & 8 & 9 & 0 & 0 \\
0 & 0 & 7 & 9 & 3 & 2 & 0 \\
0 & 0 & 0 & 3 & 8 & 4 & 6 \\
0 & 0 & 0 & 0 & 2 & 4 & 4
\end{pmatrix} \tag{2.4.4}
$$

其中 $N=7, m_1=2, m_2=1$，存储的压缩形式为 $7\times 4$ 矩阵.

$$
\begin{pmatrix}
x & x & 3 & 1 \\
x & 4 & 1 & 5 \\
9 & 2 & 6 & 5 \\
3 & 5 & 8 & 9 \\
7 & 9 & 3 & 2 \\
3 & 8 & 4 & 6 \\
2 & 4 & 4 & x
\end{pmatrix} \tag{2.4.5}
$$

这里 $x$ 表示压缩格式中浪费的空间，它们在运算中不被引用，因而其值可随意。注意，原矩阵对角线元素出现在第 $m_1+1$ 列中，对角线左下的元素在其左侧，对角线右上的元素在其右侧。

压缩存储的带状对角矩阵最简单的运算是乘以其右边的向量。尽管这是算法上的小问题，读者可能想仔细研究一下下面的程序，看看如何把压缩存储格式的非零元 $a_{ij}$ 变成顺序的方式。

```
#include "nrutil.h"

void banmul(float **a, unsigned long n, int m1, int m2, float x[], float b[])
```

矩阵乘法 **b**=**A**·**x**，其中 **A** 是带状对角阵，对角线下有 m1 行，对角线上有 m2 行。输入向量 **x** 及输出向量 **b** 相应存作 x[1..n] 和 b[1..n]。数组 a[1..n][1..m1+m2+1] 存储 **A**：对角线元素在 a[1..n][m1+1] 中，对角线左下的元素在数组 a[j..n][1..m1] 中 ($j>1$，对应于每个对角线左下元素的个数)。对角线右上元素在 a[1..j][m1+2..m1+m2+1] 中，$j<n$，对应于每个对角线右上元素的个数。

```
{
    unsigned long i,j,k,tmploop;

    for (i=1;i<=n;i++) {
        k=i-m1-1;
        tmploop=LMIN(m1+m2+1,n-k);
        b[i]=0.0;
        for (j=LMAX(1,1-k);j<=tmploop;j++) b[i] += a[i][j]*x[j+k];
    }
}
```

不可能把一个带状对角矩阵 **A** 的 $LU$ 分解也象其压缩形式本身一样紧凑地存储起来。分解(主要是因克劳特方法，参见第2.3节)产生了附加的非零元素填入。一种直接的存储方案是，把上三角因子($U$)返回到 **A** 以前占有的相同的空间中，把下三角因子($L$)返回到单独的 $N\times m_1$ 压缩矩阵中。$U$ 的对角线元素(它们的乘积，乘上 $d=\pm1$，便是行列式的值)被存放在 **A** 的存储空间的第一列。

下面的程序 **bandec** 类似于第2.3节中的 **ludcmp**，但是解带状对角系统的。

```
#include <math.h>
#define SWAP(a,b) {dum=(a);(a)=(b);(b)=dum;}
#define TINY 1.0e-20
```

```
void bandec(float **a, unsigned long n, int m1, int m2, float **al, unsigned long indx[], float
```
对给定的 n×n 带状对角矩阵 A，其对角线左下有 m1 行，对角线右上有 m2 行，被压缩存储在数组 a[1..n][1..m1+m2+1]中，见程序 **banmul** 的说明部分。本程序构造一个 A 按行排列的 LU 分解。上三角矩阵替代 a，下三角矩阵返回到 al[1..n][1..m1]中，indx[1..n]是输出向量，记录部分主元法影响的行排列情况；d 输出值为±1，由行交换次数的奇偶确定。本程序与 **banbks** 一起使用，求解带状对角方程。

```
{
    unsigned long i,j,k,l;
    int mm;
    float dum;

    mm=m1+m2+1;
    l=m1;
    for (i=1;i<=m1;i++) {                                    重新组织一下存储
        for (j=m1+2-i;j<=mm;j++) a[i][j-l]=a[i][j];
        l--;
        for (j=mm-l;j<=mm;j++) a[i][j]=0.0;
    }
    *d=1.0;
    l=m1;
    for (k=1;k<=n;k++) {                                     对每一行
        dum=a[k][1];
        i=k;
        if (l < n) l++;
        for (j=k+1;j<=l;j++) {                               寻找主元素
            if (fabs(a[j][1]) > fabs(dum)) {
                dum=a[j][1];
                i=j;
            }
        }
        indx[k]=i;
        if (dum == 0.0) a[k][1]=TINY;                        矩阵算法上是奇异的，但仍用 TINY 作为主元
                                                             继续进行（在某些应用时需要这样做）
        if (i != k) {                                        行交换
            *d = -(*d);
            for (j=1;j<=mm;j++) SWAP(a[k][j],a[i][j])
        }
        for (i=k+1;i<=l;i++) {                               做消去
            dum=a[i][1]/a[k][1];
            al[k][i-k]=dum;
            for (j=2;j<=mm;j++) a[i][j-1]=a[i][j]-dum*a[k][j];
            a[i][mm]=0.0;
        }
    }
}
```

在 **bandec** 存储限制允许范围内，有时候求主元是可能的，上面的程序利用了这种机会。一般情况下，当返回 TINY 作为 $U$ 的对角元时，则原来的矩阵（可能由于舍入误差的原因）事实上是奇异的。考虑到这点，**bandec** 某种程度上要比上面的 **tridag** 更稳健，因为 **tridag** 即使对非奇异的矩阵也可能导致算法上的失败。**bandec** 对某些效果不好的三对角系统也是有效的（取 $m_1=m_2=1$）。

一旦矩阵 A 被分解了，任意数目的右端项都可以通过重复调用 **banbks** 来求解。回代程序与第2.3节中的 **lubksb** 类似。

```
#define SWAP(a,b) {dum=(a);(a)=(b);(b)=dum;}

void banbks(float **a, unsigned long n, int m1, int m2, float **al, unsigned long indx[], float b[])
```
对给定作为 bandec 返回值的数组 a、al 和 index，及给定的右端项向量 b[1..n]，求解带状对角线性方程 A·X=b。解向量 **X** 覆盖写入 $b[1..n]$。其它的输入数组不被改变，并且可被用于以后不同右端项的连续调用。

```
{
    unsigned long i,k,l;
    int mm;
    float dum;

    mm=m1+m2+1;
    l=m1;
    for (k=1;k<=n;k++) {              前代，不打乱行的排列顺序
        i=indx[k];
        if (i != k) SWAP(b[k],b[i])
        if (l < n) l++;
        for (i=k+1;i<=l;i++) b[i] -= al[k][i-k]*b[k];
    }
    l=1;
    for (i=n;i>=1;i--) {              回代
        dum=b[i];
        for (k=2;k<=l;k++) dum -= a[i][k]*b[k+i-1];
        b[i]=dum/a[i][1];
        if (l < mm) l++;
    }
}
```

程序 **bandec** 和 **banbks** 基于参考书目[1]中的程序 ***bandet*1** 和 ***bansol*1**。

参考文献和进一步读物：


Wilkinson, J. H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag), Chapter 1/6. [1]


## 2.5 线性方程组解的迭代改进

显然，很难使线性方程组解的精度超过计算机浮点字精度。不幸的是，对于大型线性方程组，要想达到甚至是接近计算机所限定的精度也总是很难的。在直接求解的方法中，舍入误差的积累达到一定程度时，矩阵就接近奇异了。对于远非奇异的矩阵(认为如此)，却也很容易丢失两个或三个有效数字。

如果发生这种情况，有一个简洁的方法能恢复到整个机器精度，称作解的**迭代改进**。理论上是很直接的(见图2.5.1)：假定向量 $\mathbf{x}$ 是线性方程组

$$\mathbf{A}\cdot\mathbf{x}=\mathbf{b} \tag{2.5.1}$$

的精确解。然而并不知道解 $\mathbf{x}$，而仅仅知道一些略有误差的解 $\mathbf{x}+\delta\mathbf{x}$，其中 $\delta\mathbf{x}$ 为未知的误差。当将其乘以矩阵 $\mathbf{A}$ 时，这个有误差的解使乘积与原来的右端项 $\mathbf{b}$ 也存在偏差，即

$$\mathbf{A}\cdot(\mathbf{x}+\delta\mathbf{x})=\mathbf{b}+\delta\mathbf{b} \tag{2.5.2}$$

式(2.5.2)减去式(2.5.1)，得

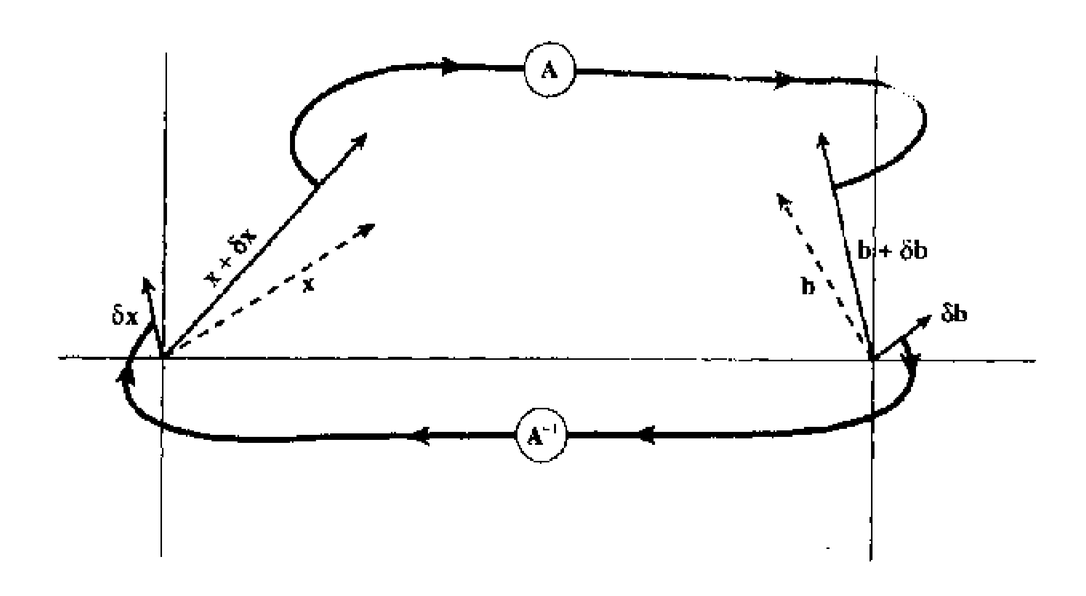
$$\mathbf{A}\cdot\delta\mathbf{x}=\delta\mathbf{b} \tag{2.5.3}$$

但由式(2.5.2)也可求得 $\delta\mathbf{b}$，将其代入式(2.5.3)中得

$$\mathbf{A}\cdot\delta\mathbf{x}=\mathbf{A}\cdot(\mathbf{x}+\delta\mathbf{x})-\mathbf{b} \tag{2.5.4}$$

该方程中，整个右端项是已知量，因为 $\mathbf{x}-\delta\mathbf{x}$ 是有待改进的有误差的解。用双精度数计算右端项是个好主意，因为在减去 $\mathbf{b}$ 的过程中会产生很多抵消。接下来我们只需求解方程(2.5.4)的偏差量 $\delta\mathbf{x}$，将其从有误差的解中减去便得到改进后的解。

如果我们是通过 *LU* 分解得到最初的解，则这么做还有一个很大的好处。因为这种情况下，已经有了矩阵 **A** 的 *LU* 分解形式，要求解方程(2.5.4)，只需计算右端项再进行回代！




首先假定 $\mathbf{x}+\delta\mathbf{x}$ 与 **A** 相乘的积为 $\mathbf{b}+\delta\mathbf{b}$，减去已知向量 **b** 得 $\delta\mathbf{b}$。再以此为右端项反过来求解该线性方程组，得 $\delta\mathbf{x}$，将其从开始假定的解中减去即得改进后的解 **x**。

图2.5.1　$\mathbf{A}\cdot\mathbf{x}=\mathbf{b}$ 的解的迭代改进

完成上述功能的代码既简洁又直接：

```
#include "nrutil.h"

void mprove(float **a, float **alud, int n, int indx[], float b[], float x[])
    改进线性方程式 A·x=B 的解向量 x[1..n]。矩阵 a[1..n][1..n]以及 n 维向量 b[1..n]和 x 均为输入量。alud[1..n][1..
    n]也为输入量，它是 ludcmp 返回的 LU 分解形式，向量 indx[1..n]也由该程序返回。输出量只有 x 被修改，它为解
    集的改进值。
{
    void lubksb(float **a, int n, int *indx, float b[]);
    int j,i;
    double sdp;
    float *r;

    r=vector(1,n);
    for (i=1;i<=n;i++) {                          计算右端项，用双精度数表示差值
        sdp = -b[i];
        for (j=1;j<=n;j++) sdp += a[i][j]*x[j];
        r[i]=sdp;
    }
    lubksb(alud,n,indx,r);                        求解的误差量 δx
    for (i=1;i<=n;i++) x[i] -= r[i];              将其从原来的解中减去
    free_vector(r,1,n);
}
```

应当注意第2.3节中的程序 **ludcmp** 破坏了输入矩阵，将其进行了 *LU* 分解。因为迭代法既需要原来的矩阵又需要其 *LU* 分解的形式，在调用 **ludcmp** 之前须将 **A** 进行复制。同理

lubksb 破坏了 b 而得到 x，所以要复制一份 b。如果不在意这些额外增加的存储空间，则迭代改进法是很值得推荐的；其运算量仅为 $N^2$级（向量矩阵乘及回代，见方程(2.3.7)下面的讨论），而且它永远不会把解变糟。如果已经花了 $N^3$次运算，而得到的却是一个不够精确的解，则这个迭代算法确实是很划得来的。

如果愿意，可以连续调用 mprove 好几次。除非刚开始的解与真解相差太远，一般来说一次调用就足够了。但也可以再调用一次来证实其收敛性。

## 2.5.1 关于迭代改进的更多的讨论

对方程(2.5.4)给出略微更可靠些的分析是有启发意义的（而且本书后面还要用到），并且还能另外得到一些结果。在前面的讨论中，已指出解向量 $\mathbf{x}+\delta\mathbf{x}$ 有一个错误的项，但我们却忽略了一个事实，即 A 的 $LU$ 分解本身并不精确。

一种不同的分析方法由矩阵 $\mathbf{B}_0$开始，假定它是矩阵 A 的一个近似的逆，则 $\mathbf{B}_0\cdot\mathbf{A}$ 近似等于单位矩阵 1。定义 $\mathbf{B}_0$的剩余矩阵 R 为：

$$\mathbf{R} \equiv \mathbf{1} - \mathbf{B}_0 \cdot \mathbf{A} \tag{2.5.5}$$

其值假定都“很小”（下面会给出精确的说明）。因此，

$$\mathbf{B}_0 \cdot \mathbf{A} = \mathbf{1} - \mathbf{R} \tag{2.5.6}$$

接下来考虑下面的标准运算：

$$\begin{aligned}\mathbf{A}^{-1} &= \mathbf{A}^{-1} \cdot (\mathbf{B}_0^{-1} \cdot \mathbf{B}_0) = (\mathbf{A}^{-1} \cdot \mathbf{B}_0^{-1}) \cdot \mathbf{B}_0 \\ &= (\mathbf{B}_0 \cdot \mathbf{A})^{-1} \cdot \mathbf{B}_0 = (\mathbf{1} - \mathbf{R})^{-1} \cdot \mathbf{B}_0 = (\mathbf{1} + \mathbf{R} + \mathbf{R}^2 + \mathbf{R}^3 + \cdots) \cdot \mathbf{B}_0\end{aligned} \tag{2.5.7}$$

我们可以将最后一个表达式的前 n 项之和定义为

$$\mathbf{B}_n \equiv (\mathbf{1} + \mathbf{R} + \cdots \mathbf{R}^n) \cdot \mathbf{B}_0 \tag{2.5.8}$$

因此，如果极限存在，则 $\mathbf{B}_\infty \rightarrow \mathbf{A}^{-1}$。

现在，直接来验证一下等式(2.5.8)满足一些有趣的递归关系。考虑到要求解方程组 $\mathbf{A}\cdot\mathbf{x}=\mathbf{b}$，其中 x 和 b 都是向量，定义

$$\mathbf{x}_n \equiv \mathbf{B}_n \cdot \mathbf{b} \tag{2.5.9}$$

则很容易得到

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{B}_0 \cdot (\mathbf{b} - \mathbf{A} \cdot \mathbf{x}_n) \tag{2.5.10}$$

立刻可以看出，它跟方程(2.5.4)一样，其中 $-\delta\mathbf{x}=\mathbf{x}_{n+1}-\mathbf{x}_n$，$\mathbf{B}_0$这里代替了 $\mathbf{A}^{-1}$的位置。因此，我们看到方程(2.5.4)并不需要 A 的 $LU$ 分解有多么精确，而只是要求剩余矩阵 R 很小即可。粗略而言，如果该剩余量小于计算机舍入误差的平方根，则使用一次方程(2.5.10)之后（即由 $\mathbf{x}_0\equiv\mathbf{B}_0\cdot\mathbf{b}$ 得到 $\mathbf{x}_1$），第一次忽略的项，为 $\mathbf{R}^2$数量级，也就小于舍入误差了。而且，方程(2.5.10)与(2.5.4)一样可以使用多次，因为它仅仅使用了 $\mathbf{B}_0$，而没用更高阶的 $\mathbf{B}_n$。

由等式(2.5.8)可以得到一个更令人吃惊的递推式，它每次递推的阶 $n$ 要超过2倍：

$$\mathbf{B}_{2n+1} = 2\mathbf{B}_n - \mathbf{B}_n \cdot \mathbf{A} \cdot \mathbf{B}_n \qquad n = 0,1,3,7,\cdots \tag{2.5.11}$$

从一个合适的矩阵 $\mathbf{B}_0$开始，重复使用等式(2.5.11)，可以二次地收敛于未知的逆矩阵 $\mathbf{A}^{-1}$（“二次地”定义参见第9.4节），等式(2.5.11)有许多不同的名字，包括 Schultz 方法和 Hotelling 方法，参见 Pan 和 Reif[1]。事实上，等式(2.5.11)只不过是找根的迭代 Newton-Raphson 方法应用于矩阵求逆罢了。

然后，不要对等式(2.5.11)过于乐观，应当注意到每次迭代它要进行两次完全的矩阵乘法。每次矩阵相乘包括 $N^3$加法和乘法。但是，我们在第2.1～2.3节的 *toto* 中，已经知道直接求 A 的逆也只需要 $N^3$次加法和 $N^3$次乘法。因此等式(2.5.11)仅在某些特殊情况下，其运算要比通用矩阵求解方法快时才有意义，这些情况我们将在后面的第13.10节中遇到。

我们暂且来研究一下两个相关的问题：什么时候等式(2.5.7)的级数收敛；什么样的初始假设 $\mathbf{B}$ 合适(例如，如果初始的 $LU$ 分解不可行)？

我们可以将矩阵的**范数**(*norm*)定义为，它能对一个向量长度放大的最大倍数，

$$\|\mathbf{R}\| \equiv \max_{\mathbf{v} \neq 0} \frac{|\mathbf{R} \cdot \mathbf{v}|}{|\mathbf{v}|} \tag{2.5.12}$$

如果使方程(2.5.7)取任意的右端项 $\mathbf{b}$，即需要矩阵的逆时，显然收敛的充分条件是

$$\|\mathbf{R}\| < 1 \tag{2.5.13}$$

Pan 和 Reif[1]中指出，一个合适的 $\mathbf{B}_0$ 的初始假设是，充分小的常量 $\epsilon$ 乘以 $\mathbf{A}$ 的转置，即

$$\mathbf{B}_0 = \epsilon \mathbf{A}^T \qquad \text{或} \qquad \mathbf{R} = \mathbf{1} - \epsilon \mathbf{A}^T \cdot \mathbf{A} \tag{2.5.14}$$

要搞清楚为什么这里使用了第11章的概念，我们仅给出最简单的介绍：$\mathbf{A}^T \cdot \mathbf{A}$ 是对称的、正定矩阵，故它有正实数的特征值，用其对角线表示，$\mathbf{R}$ 采用的形式为

$$\mathbf{R} = \mathrm{diag}(1 - \epsilon\lambda_1, 1 - \epsilon\lambda_2, \cdots, 1 - \epsilon\lambda_N) \tag{2.5.15}$$

其中所有的 $\lambda_i$ 均为正数。显然，对任何 $\epsilon$ 满足 $0 < \epsilon < 2/(\max_i \lambda_i)$ 都得到 $\|\mathbf{R}\| < 1$。不难发现，$\epsilon$ 的最优选择给出等式(2.5.11)最快的收敛，它是

$$\epsilon = 2/(\max_i \lambda_i + \min_i \lambda_i) \tag{2.5.16}$$

人们很少知道等式(2.5.16)中 $\mathbf{A}^T \cdot \mathbf{A}$ 的特征值。Pan 和 Reif[1]得到了几个有趣的边界，它们可以由 $\mathbf{A}$ 直接计算出来。下面的选择保证了 $n \to \infty$ 时，$\mathbf{B}_n$ 的收敛性

$$\epsilon \leqslant 1 / \sum_{j,k} a_{jk}^2 \qquad \text{或者} \qquad \epsilon \leqslant 1 / \left(\max_i \sum_j |a_{ij}| \times \max_j \sum_i |a_{ij}|\right) \tag{2.5.17}$$

第二个的表达是真正值得注意的式子，由 Pan 和 Reif 推得，他们注意到，等式(2.5.12)中向量的范数不必是通常 $L_2$ 的范数，但可被 $L_\infty$(最大值)范数或是 $L_1$(绝对值)范数所代替，详见他们的著作。

另外一种方法我们也获得了一些成功，就是通过在 $N$ 维空间中随机选取的方向上，选几个单位向量 $\mathbf{v}_i$ 来计算 $s_i = |\mathbf{A} \cdot \mathbf{v}_i|^2$，用统计的方法来估算最大的特征值。最大的特征值 $\lambda$ 可以被 $2\max s_i$ 及 $2N\mathrm{Var}(s_i)/\mu(s_i)$ 的最大值所限定，其中 Var 和 $\mu$ 相应表示样本的方差和均值。

参考文献及进一步读物：


Pan, V., and Reif, J. 1985, in Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing (New York: Association for Computing Machinery). [1]


## 2.6 奇异值分解

有一套非常有用的技术，来处理不论是奇异的，还是数值上非常接近奇异的方程组或矩阵。在许多情况下，高斯消去法和 $LU$ 分解法都不能给出满意的结果，这套称为**奇异值分解**或 SVD 的技术，会精确地告诉我们什么样的情况属于该类问题。某些情况下，SVD 不仅仅可以判断该问题，也可以求解它，向读者提供一个有用的数值解，尽管我们将会看到这个解也不一定是读者想要得到的。

SVD 法也可用来求解大多数的**线性最小二乘方**问题，这一节简要地介绍一下相关的理论，有关 SVD 在这方面应用的详细探讨放在第十五章中，那一章的专题即为数据模型法。

SVD 法是基于下面的线性代数定理，其证明超出了我们的范围：任意 $M \times N$ 的矩阵 $\mathbf{A}$，其行数 $M$ 大于或等于列数 $N$，可以写成一个 $M \times N$ 的列正交矩阵 $\mathbf{U}$，一个 $N \times N$ 的元素均

为正数或零的对角阵 $W$、以及一个 $N\times N$ 正交阵 $\mathbf{V}$ 的转置矩阵的乘积形式。用下图表示这些矩阵的各种形式会更清楚些：

$$\left(\mathbf{A}\right)=\left(\mathbf{U}\right)\cdot\begin{pmatrix} w_1 & & & \\ & w_2 & & \\ & & \cdots & \\ & & & w_N \end{pmatrix}\cdot\left(\mathbf{V}^T\right) \tag{2.6.1}$$

矩阵 **U** 和 **V** 都是正交的，它们的列是正交标准化的

$$\sum_{i=1}^{M} U_{ik}U_{in}=\delta_{kn} \qquad 1\leqslant k\leqslant N \tag{2.6.2}$$

$$1\leqslant n\leqslant N$$

$$\sum_{j=1}^{N} V_{jk}V_{jn}=\delta_{kn} \qquad 1\leqslant k\leqslant N \tag{2.6.3}$$

$$1\leqslant n\leqslant N$$

或者用下图来表示：

$$\left(\mathbf{U}^T\right)\cdot\left(\mathbf{U}\right)=\left(\mathbf{V}^T\right)\cdot\left(\mathbf{V}\right)=\left(\mathbf{1}\right) \tag{2.6.4}$$

由于 **V** 是方阵，故也是行正交标准化的，$\mathbf{V}\cdot\mathbf{V}^T=1$。

当 $M<N$ 时，SVD 分解也可以执行。在这情况中，奇异值 $w_j$，$j=M-1,\cdots,N$ 都等于零，并且 **U** 中相应的列都是零。这时，仅对 k，n$\leqslant M$ 时，式(2.6.2)成立。

式(2.6.1)的分解总是可以完成的，而不管矩阵是否是奇异的，而且这个分解几乎是唯一的。也就是说，其分解形式唯一到(i)对 **U** 的列，**W** 的元素和 **V** 的列(或 $\mathbf{V}^T$ 的行)能做相同的置换，或者(ii)形成 **U** 和 **V** 的任意列的线性组合，其在 **W** 中对应的元素仍恰好完全相同。对于 $M<N$ 的情况，这种自由置换的重要结论是，分解的数值算法对 $w_j$，$j=M+1,\cdots,N$ 要求返回非零，而这 $N-M$ 个零奇异值应分散于所有 $j=1,2,\cdots,N$ 的位置上。

在本节的最后，我们给出一个程序 **svdcmp**，对任意矩阵 **A** 进行 SVD 变换，将其替换成 **U**(它们的形式是相同的)并且分别给出 **W** 和 **V**。程序 **svdcmp** 是基于 Forsythe 等人提供的程序，而 Forsythe 等人的这个程序又是基于 Golub 和 Reinsch 原来的程序，在参考书[2]—[4]以及其它一些地方都可以找到它的变形。这些参考书中，还包括对所用算法进行的广泛的讨论。尽管我们也不喜欢使用黑箱程序，但我们还是要求读者接受它，因为这样就不必花太多功夫去研究必要的背景知识了。有充足的理由表明，该算法是非常稳定的，出现失误是极例外的情况。关于该算法的大部分概念，(普通矩阵化简成双对角矩阵的形式，带变换的 QR 过程进行对角线化)，将在第十一章中进行讨论。

如果读者跟我们一样对黑箱产生怀疑，你会想亲自验证一下 **svdcmp** 是否完成我们所说的功能。这很容易做到：任意生成一个矩阵 **A**，调用该程序，然后验证一下矩阵的乘积是否满足式(2.6.1)和式(2.6.4)。由于这两个方程是定义 SVD 方法的唯一限定条件，这样做(对所选定的 **A**)便是完全彻头彻尾的验证了。

现在我们来看看 SVD 适合做些什么。

### 2.6.1 方阵的 SVD

如果矩阵 **A** 为方阵，假设为 $N \times N$ 阶的，则 **U**、**V**、和 **W** 均为相同大小的方阵。它们的逆也很容易计算：**U** 和 **V** 是正交的，因而它们的逆就是它们的转置；**W** 为对角阵，它的逆也是对角阵，其元素为原来元素 $w_j$ 的倒数。由式(2.6.1)可直接得到 **A** 的逆为

$$\mathbf{A}^{-1} = \mathbf{V} \cdot [\mathrm{diag}(1/w_j)] \cdot \mathbf{U}^T \tag{2.6.5}$$

这么做唯一可以出错的地方是 $w_j$ 中有一个为零，或其(数值上)非常小，由于舍入误差导致丢失。如果不只一个的 $w_j$ 存在这种问题，则矩阵就更加奇异。因此，SVD 首先就让使用者清楚地知道这种情况。

形式上，矩阵的**条件数**定义为 $w_j$ 的最大元与最小元的比值。如果一个矩阵的条件数无穷大，则该矩阵是奇异的；如果一个矩阵的条件数太大，即其倒数超出了机器的浮点精度(比如，单精度数小于$10^{-6}$或双精度数小于$10^{-12}$)，则称该矩阵是**"病态的"**。

对奇异矩阵，**零空间**和**值域**的概念是很重要的。考虑下面的联立方程式

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.6.6}$$

其中 **A** 为方阵，**b** 和 **x** 为向量。方程(2.6.6)将 **A** 定义为从向量空间 **x** 到向量空间 **b** 的一个线性映射。如果 **A** 是奇异的，则存在子空间 **x**，称之为零空间，它被映射为零，$\mathbf{A} \cdot \mathbf{x} = 0$。零空间的维数(可在 **x** 中找到的非线性依赖的向量个数)称为 **A** 的**零维数**。

还有一些通过 **A** 可以"到达"的子空间 **b**，就是说存在一些 **x** 被映射到其上。子空间 **b** 被称为 **A** 的值域。值域的维数称为 **A** 的**秩**。如果 **A** 为非奇异的，则其值域为整个向量空间 **b**，即其秩为 $N$。如果 **A** 是奇异的，则秩小于 $N$。事实上，相应的定理为"秩与零维数之和等于 $N$"。

用 SVD 究竟做什么呢？SVD 明确地构造了矩阵零空间和值域的正交标准化基。特别地，对 **U** 的列，若与其标号相同的元素 $w_j$ 为非零元，则其是值域的一个正交标准化的基础向量；对 **V** 的列，若与其标号相同的 $w_j$ 为零，则其是零空间的一个正交标准化基。

现在，再看一下 **A** 为奇异的情况下，联立线性方程组(2.6.6)的求解过程。首先，对于齐次线性方程组，因其 $\mathbf{b}=0$，所以可以立刻用 SVD 方法求解：对应于 $w_j$ 为零的 **V** 中的某列产生一个解。

当右端项向量 **b** 不为零时，重要的问题是，对右端项向量 **b** 是否存在 **A** 的值域。如果存在，则该奇异矩阵确实有解；事实上解不止一个，因为零空间的任意向量(**V** 中对应 $w_j$ 为零的列)进行任意线性组合都可以加到 **x** 上。

如果我们仅想求出该解向量集合中一个特殊的有代表性的解，则我们可以求具有最小长度 $|\mathbf{x}|^2$的那个解。下面介绍使用 SVD 法求该向量：如果 $w_j=0$，简单地将$1/w_j$ 替换成零即可。(设置$\infty=0$的情况并不常见！)然后计算(从右向左)

$$\mathbf{x} = \mathbf{V} \cdot [\mathrm{diag}(1/w_j)] \cdot (\mathbf{U}^T \cdot \mathbf{b}) \tag{2.6.7}$$

这就是具有最小长度的解向量，在零空间中 V 的列完成对解集的要求。

证明：考虑 $|\mathbf{x}+\mathbf{x}'|$，其中 $\mathbf{x}'$ 属于零空间。设 $\mathbf{W}^{-1}$ 表示有零元素的矩阵 $\mathbf{W}$ 被修改过后的逆矩阵。

$$
\begin{aligned}
|\mathbf{x}+\mathbf{x}'| &= |\mathbf{V}\cdot\mathbf{W}^{-1}\cdot\mathbf{U}^T\cdot\mathbf{b}+\mathbf{x}'| \\
&= |\mathbf{V}\cdot(\mathbf{W}^{-1}\cdot\mathbf{U}^T\cdot\mathbf{b}+\mathbf{V}^T\cdot\mathbf{x}')| \\
&= |\mathbf{W}^{-1}\cdot\mathbf{U}^T\cdot\mathbf{b}+\mathbf{V}^T\cdot\mathbf{x}'|
\end{aligned}
\quad (2.6.8)
$$

其中第一个等式由(2.6.7)导出，第二个和第三个由 V 是正交标准化得出。如果检验一下右端求和的两项，会发现第一项中仅当 $w_j\neq 0$ 时，第 $j$ 个分量为非零；而对于第二项，由于 $\mathbf{x}'$ 在零空间中，仅当 $w_j=0$ 时，第 $j$ 个分量为非零。故对 $\mathbf{x}'=0$时，其长度最小，证毕。

如果 $\mathbf{b}$ 不在奇异矩阵 $\mathbf{A}$ 的值域范围内，则方程组(2.6.6)无解。但值得高兴的是：如果 $\mathbf{b}$ 不在 $\mathbf{A}$ 的值域中，方程组(2.6.7)还是可以构造一个“解”向量 $\mathbf{x}$ 的。该向量并不是 $\mathbf{A}\cdot\mathbf{x}=\mathbf{b}$ 的精确解，但在所有可能的向量 $\mathbf{x}$ 中，它是在最小二乘意义上最接近的解。换言之，方程(2.6.7)求得的 $\mathbf{x}$，使

$$
r\equiv|\mathbf{A}\cdot\mathbf{x}-\mathbf{b}| \quad (2.6.9)
$$

最小。数 $r$ 被称为解的**残差**。

证明与式(2.6.8)类似：假设对 $\mathbf{x}$ 加上了某个任意的 $\mathbf{x}'$，则 $\mathbf{A}\cdot\mathbf{x}-\mathbf{b}$ 也加上了一项 $\mathbf{b}'=\mathbf{A}\cdot\mathbf{x}'$。显然 $\mathbf{b}'$ 在 $\mathbf{A}$ 的值域中，则有

$$
\begin{aligned}
|\mathbf{A}\cdot\mathbf{x}-\mathbf{b}+\mathbf{b}'| &= |(\mathbf{U}\cdot\mathbf{W}\cdot\mathbf{V}^T)\cdot(\mathbf{V}\cdot\mathbf{W}^{-1}\cdot\mathbf{U}^T\cdot\mathbf{b})-\mathbf{b}+\mathbf{b}'| \\
&= |(\mathbf{U}\cdot\mathbf{W}\cdot\mathbf{W}^{-1}\cdot\mathbf{U}^T-1)\cdot\mathbf{b}+\mathbf{b}'| \\
&= |\mathbf{U}\cdot[(\mathbf{W}\cdot\mathbf{W}^{-1}-1)\cdot\mathbf{U}^T\cdot\mathbf{b}+\mathbf{U}^T\cdot\mathbf{b}']| \\
&= |(\mathbf{W}\cdot\mathbf{W}^{-1}-1)\cdot\mathbf{U}^T\cdot\mathbf{b}-\mathbf{U}^T\cdot\mathbf{b}'|
\end{aligned}
\quad (2.6.10)
$$

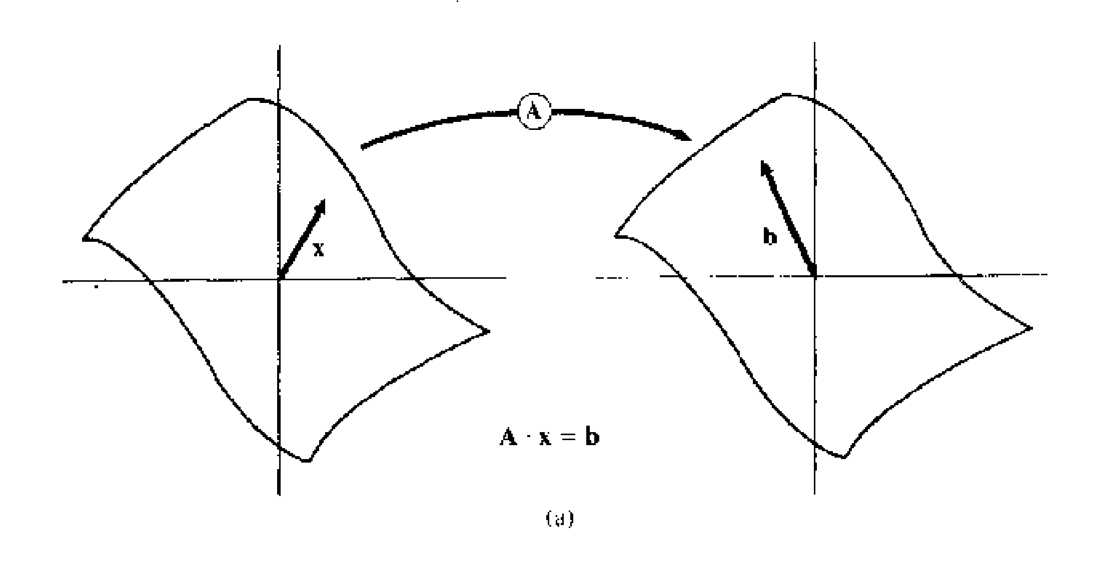
$(\mathbf{W}\cdot\mathbf{W}^{-1}-1)$为对角阵，只有当 $w_j=0$ 时其第 $j$ 个分量为非零。而由于 $\mathbf{b}'$ 为 $\mathbf{A}$ 的值域，故只有当 $w_j\neq 0$ 时，$\mathbf{U}^T\mathbf{b}'$ 的第 $j$ 个分量为非零。因此，令 $\mathbf{b}'=0$ 即得最小值，证毕。

图2.6.1是对所讨论 SVD 法的总结。

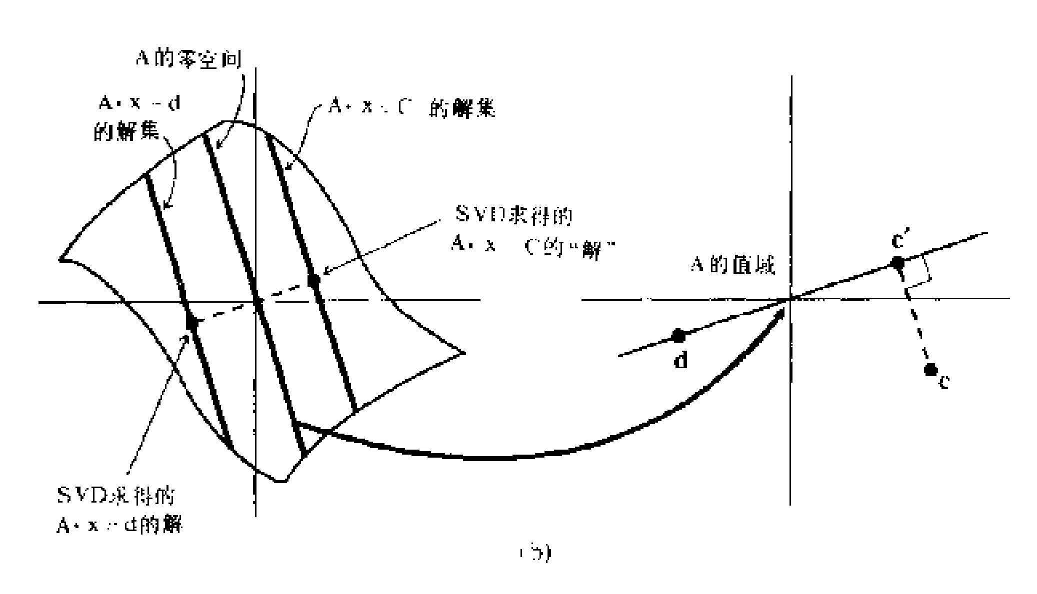
从方程(2.6.6)以来的讨论中，都已经假定不管矩阵是奇异的还是非奇异的，这从分析上看自然是对的。然而在数值上看，更一般的情况为，某些 $w_j$ 非常小但不为零，即矩阵是“病态”的。在这种情况下，$LU$ 分解法或高斯消去法等直接解法，可能给出方程组的形式解(即没遇到零主元素)。但解向量可能有非常大的分量，当与矩阵 $\mathbf{A}$ 相乘时，由于代数上的相互抵消而得到右端项向量 $\mathbf{b}$ 的一个非常糟糕的近似解。遇到这些情况时，将这些小的 $w_j$ 化为零，再用方程(2.6.7)来求解，求得的解向量 $\mathbf{x}$ 比用直接法和保留这些非零的小 $w_j$ 的 SVD 解法都要好得多(在使残差$|\mathbf{A}\cdot\mathbf{x}-\mathbf{b}|$尽可能小的意义上看)。

看起来这可能有点矛盾，因为，将奇异数值化为零相当于把要求解的方程组所生成的一个线性组合抛弃。矛盾的解决在于我们抛弃的这个方程的线性组合由于舍入误差已变得非常糟。最好的情况也是无用的，而通常比无用还要糟，因为它将差不多已是零空间向量的解沿某个方向“拖”偏了一点。这样，它就掺进了舍入误差而使得残差$|\mathbf{A}\cdot\mathbf{x}-\mathbf{b}|$更大了。

SVD 方法不能盲目使用。必须谨慎地确定将小 $w_j$ 化为零的阈值，而且(或者)必须考虑清楚多大的计算残差$|\mathbf{A}\cdot\mathbf{x}-\mathbf{b}|$可以接受。




(a)




(b)

(a)非奇异矩阵A将一个向量空间映射到一个具有相同维数的向量空间，向量x被映射成b，因此x满足方程A·x=b。(b)奇异矩阵A将一个向量空间映射到低维空间，这里用一个平面映射到一条直线表示，该直线称作A的"值域"。A的"零空间"被映射成零。A·x=d的解集包含任意一个特殊解加上零空间的任意向量，这里用平行于零空间的直线表示。奇异值分解(SVD)选择离零最近的特殊解，如图示。点C不在A的值域范围内，所以A·x=c无解。SVD找到最小二乘意义上的最优的折衷解，即A·x=c′的解，如图示。

图2.6.1

作为示例，有一个"回代"程序 **svbskb** 用来评价方程(2.6.7)，并得到右端项 **b** 的一个解向量 **x**，要求先调用 **svdcmp** 计算出矩阵 **A** 的 SVD 形式。注意该程序假定已经将小的 $w_j$ 化为零了，因该程序不提供此项功能。如果还没有将小 $W_j$ 零化，则该程序跟别的直接方法一样是"病态"的，这样就误用了 SVD 法。

```
#include "nrutil.h"

void svbksb (float **u, float w[], float **v, int m, int n, float b[], float x[])
```

求 **A·x=B** 的解向量 **x**，其中 **A** 由 **svdcmp** 返回的数组 u[1..m][1..n]，w[1..n] 和 v[1..n][1..n] 确定，m 和 n 是 **A** 的维数，对方程来说二者相同。b[1..m] 为输入右端项，x[1..n] 为输出的解向量。输入量不被破坏，因此程序可以用不同的 **b** 顺序调用

```
{
    int jj,j,i;
    float s,*tmp;

    tmp=vector(1,n);
    for (j=1;j<=n;j++) {          计算U^T B
        s=0.0;
        if (w[j]) {               仅当W_j为非零时，结果为非零
            for (i=1;i<=m;i++) s += u[i][j]*b[i];
            s /= w[j];            这里用W_j除
        }
        tmp[j]=s;
    }
    for (j=1;j<=n;j++) {          将矩阵乘以V以得到解
        s=0.0;
        for (jj=1;jj<=n;jj++) s += v[j][jj]*tmp[jj];
        x[j]=s;
    }
    free_vector(tmp,1,n);
}
```

注意，**svdcmp** 和 **svbksb** 的典型用法表面上跟 **ludcmp** 和 **lubksb** 的典型用法很相象：这两种情况下，都将左端矩阵 **A** 仅分解一次，然后可以对右端项使用该分解一次，或是用不同的右端项使用该分解多次。最重要的不同之处在于，在调用 **svbksb** 之前要对奇异数值进行一下"编译"：

```
#define N ...
float wmax,wmin,**a,**u,*w,**v,*b,*x;
int i,j;
...
for(i=1;i<=N;i++)                                  如果不想破坏a,将其副本存入u
    for j=1;j<=N;j++)
        u[i][j]=a[i][j];
svdcmp(u,N,N,w,v);                                 对方阵a进行SVD
wmax=0.0;
for(j=1;j<=N;j++) if (w[j] > wmax) wmax=w[j];
    wmin=wmax*1.0e-6;                              设置的允许奇异值为非零的阈值,此常数是
                                                   典型值,但不是一成不变的,可根据应用条件
                                                   试着改动
for(j=1;j<=N;j++) if (w[j] < wmin) w[j]=0.0;
svbksb(u,w,v,N,N,b,x);                             现在可以回代了
```

### 2.6.2 方程数少于未知数的 SVD

如果方程数 $M$ 少于未知数 $N$，则不能指望得到唯一解了。通常存在一个 $N-M$ 维的解系。如果要找到整个的解空间，则 SVD 可以完成这项工作。

因为 $M<N$，所以 SVD 分解将产生 $N-M$ 个零或可忽略不计的 $w_j$。$M$ 个方程中的退化情况，还会产生额外的零 $w_j$。在调用 **svbksb** 之前，一定要保证找到这些小的 $w_j$，并将其化为零，则可以得到特殊的解向量 **x**。跟以前一样，**V** 中对应于零化 $w_j$ 的列是基础解向量，其线性组合加上特定的解就构成一个解空间。

### 2.6.3 方程数多于未知数的 SVD

这种情形将在第十五章中遇到，对超定的线性方程组，我们希望找到的是其最小二乘解。用图表示，要求解的方程组为

$$\begin{bmatrix} & \mathbf{A} & \end{bmatrix} \cdot \begin{bmatrix} \mathbf{x} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \end{bmatrix} \tag{2.6.11}$$

在上面方阵情况下所给出的证明，不用修改就适用于方程组多于未知数的情况。最小二乘解向量 x 由式(2.6.7)给出，对非方阵的情况如下，

$$\begin{bmatrix} \mathbf{x} \end{bmatrix} = \begin{bmatrix} & \mathbf{V} & \end{bmatrix} \cdot \begin{bmatrix} \mathrm{diag}(1/w_j) \end{bmatrix} \cdot \begin{bmatrix} & \mathbf{U}^T & \end{bmatrix} \cdot \begin{bmatrix} \mathbf{b} \end{bmatrix} \tag{2.6.12}$$

一般情况下，矩阵 **W** 不是奇异的，就没有 $w_j$ 要被设为零。然而，偶而 **A** 也可能是列退化的。这种情况下，就需要将小的 $w_j$ 值化成零了。**V** 中相应列给出的 x 的线性组合，这时即使假定是超定方程组来说，其限定条件也要变糟。

有时候，尽管由于计算上的原因不必零化任何 $w_j$，然而，读者可能会注意那些特别小的值：它们在 **V** 中对应的列是 **x** 的线性组合，但它们对读者的数据并不敏感。事实上，读者可能想要零化这些 $w_j$，以减少其自由参量的数目。这些问题将在第十五章中，作进一步的讨论。

### 2.6.4 构造正交标准基

假设在 $M$ 维向量空间有 $N$ 个向量，$N \leqslant M$，则 $N$ 个向量跨越了整个向量空间的一些子空间。我们经常想要构造跨越相同子空间的 $N$ 个向量的一个正交标准基。教科书中的作法是，通过 Gram-Schmidt 正交化，由一个向量开始，然后每次扩展一维子空间。然而从数值上看，由于其内在的舍入误差，单纯的 Gram－Schmidt 正交化效果很差。

对一个子空间，构造正交标准基的正确办法是，通过 SVD 方法：构造一个 $M \times N$ 的矩阵 **A**，其 $N$ 列是已有的向量。将该矩阵交给 **svdcmp** 运行一遍，则矩阵 **U**（事实上 **svdcmp** 输出的 **U** 替代了 **A**）的列就是欲求的正交标准基向量。

也许还应检验一下输出的 $w_j$ 中是否含有零值。如果有，则事实上跨越的子空间不是 $N$ 维的；U 中对应于 $w_j$ 为零的列应从正交标准基中去除。

（第2.10节中讨论的 $QR$ 因式分解也可构造一个正交标准基，见 [5]）。

### 2.6.5 矩阵的近似

注意方程(2.6.1)可以重写一下，将任意矩阵 $A_{ij}$ 表示成 **U** 的列和 $\mathbf{V}^T$ 的行的外积之和的形式，奇异数值 $w_j$ 作为"权因子"，

$$A_{ij} = \sum_{k=1}^{N} w_k U_{ik} V_{jk} \tag{2.6.13}$$

如果碰到矩阵的 **A** 大部分奇异值 $w_j$ 都非常小的情况，则仅用(2.6.13)和式的一小部分项，就可得到 **A** 的很不错的估计值。这意味着仅需保存少量的 **U** 和 **V** 的列（相同的 $k$ 列）就可以恢复整个矩阵，且精度很高。同时注意到用此近似的矩阵去乘向量 **x**，也是非常高效的：可将 **x** 与每个存储的 **V** 的列做点积，并乘以相应的比例值 $w_k$，然后分别乘以对应的 **U**

的列，并求累加和。如果矩阵用很少的 $K$ 个奇异值来近似，则计算 $\mathbf{A}\cdot\mathbf{x}$ 仅需乘大约 $K(M-N)$次，而不必象求整个矩阵时用 $MN$ 次。

## 2.6.6 SVD 算法

下面是用于构造任意矩阵的奇异值分解的程序。有关方法的讨论参见第11.2节～第11.3节和参考文献[4~5]。

```
#include <math.h>
#include "nrutil.h"

void svdcmp(float **a, int m, int n, float w[], float **v)
```

给定矩阵a[1..m][1..n]，本程序求其奇异值分解 $A=U\cdot W\cdot V^T$。输出矩阵 $U$ 保存在数组a中。由奇异值构成的对角阵 $W$ 作为输出量用 w[1..n]表示，输出矩阵 $V$（不是其转置形式 $V^T$）用 v[1..n][1..n]表示。

```
{
    float pythag(float a, float b);
    int flag,i,its,j,jj,k,l,nm;
    float anorm,c,f,g,h,s,scale,x,y,z,*rv1;

    rv1=vector(1,n);
    g=scale=anorm=0.0;                          归并成双对角形式
    for (i=1;i<=n;i++) {
        l=i+1;
        rv1[i]=scale*g;
        g=s=scale=0.0;
        if (i <= m) {
            for (k=i;k<=m;k++) scale += fabs(a[k][i]);
            if (scale) {
                for (k=i;k<=m;k++) {
                    a[k][i] /= scale;
                    s += a[k][i]*a[k][i];
                }
                f=a[i][i];
                g = -SIGN(sqrt(s),f);
                h=f*g-s;
                a[i][i]=f-g;
                for (j=l;j<=n;j++) {
                    for (s=0.0,k=i;k<=m;k++) s += a[k][i]*a[k][j];
                    f=s/h;
                    for (k=i;k<=m;k++) a[k][j] += f*a[k][i];
                }
                for (k=i;k<=m;k++) a[k][i] *= scale;
            }
        }
        w[i]=scale *g;
        g=s=scale=0.0;
        if (i <= m && i != n) {
            for (k=l;k<=n;k++) scale += fabs(a[i][k]);
            if (scale) {
            for (k=l;k<=n;k++) {
                a[i][k] /= scale;
                s += a[i][k]*a[i][k];
            }
            f=a[i][l];
            g = -SIGN(sqrt(s),f);
            h=f*g-s;
            a[i][l]=f-g;
            for (k=l;k<=n;k++) rv1[k]=a[i][k]/h;
            for (j=l;j<=m;j++) {
                for (s=0.0,k=l;k<=n;k++) s += a[j][k]*a[i][k];
                for (k=l;k<=n;k++) a[j][k] += s*rv1[k];
            }
```

```
            for (k=1;k<=n;k++) a[i][k] *= scale;
        }
    }
    anorm=FMAX(anorm,(fabs(w[i])+fabs(rv1[i])));
}
for (i=n;i>=1;i--) {                        求右端变换累加和
    if (i < n) {
        if (g) {
            for (j=l;j<=n;j++)              用两次除法避免可能的下溢
                v[j][i]=(a[i][j]/a[i][l])/g;
            for (j=l;j<=n;j++) {
                for (s=0.0,k=l;k<=n;k++) s += a[i][k]*v[k][j];
                for (k=l;k<=n;k++) v[k][j] += s*v[k][i];
            }
        }
        for (j=l;j<=n;j++) v[i][j]=v[j][i]=0.0;
    }
    v[i][i]=1.0;
    g=rv1[i];
    l=i;
}
for (i=IMIN(m,n);i>=1;i--) {                求左端变换累加和
    l=i+1;
    g=w[i];
    for (j=l;j<=n;j++) a[i][j]=0.0;
    if (g) {
        g=1.0/g;
        for (j=l;j<=n;j++) {
            for (s=0.0,k=l;k<=m;k++) s += a[k][i]*a[k][j];
            f=(s/a[i][i])*g;
            for (k=i;k<=m;k++) a[k][j] += f*a[k][i];
        }
        for (j=i;j<=m;j++) a[j][i] *= g;
    } else for (j=i;j<=m;j++) a[j][i]=0.0;
    ++a[i][i];
}
for (k=n;k>=1;k--) {                        双对角阵对角化：对奇异数值循环
    for (its=1;its<=30;its++) {                             对所要作的迭代循环
        flag=1;
        for (l=k;l>=1;l--) {                做分解测试
            nm=l-1;                         注意rv1[1]总为零
            if ((float)(fabs(rv1[l])+anorm) == anorm) {
                flag=0;
                break;
            }
            if ((float)(fabs(w[nm])+anorm) == anorm) break;
        }
        if (flag) {
            c=0.0;                          如果l>1则消去rv1[l]
            s=1.0;
            for (i=l;i<=k;i++) {
                f=s*rv1[i];
                rv1[i]=c*rv1[i];
                if ((float)(fabs(f)+anorm) == anorm) break;
                g=w[i];
                h=pythag(f,g);
                w[i]=h;
                h=1.0/h;
                c=g*h;
                s = -f*h;
                for (j=1;j<=m;j++) {
                    y=a[j][nm];
                    z=a[j][i];
                    a[j][nm]=y*c+z*s;
                    a[j][i]=z*c-y*s;
                }
```

```
            }
        }
        z=w[k];
        if (l == k) {                     收敛
            if (z < 0.0) {                使奇偶数值非负
                w[k] = -z;
                for (j=1;j<=n;j++) v[j][k] = -v[j][k];
            }
            break;
        }
        if (its == 30) nrerror("no convergence in 30 svdcmp iterations");
        x=w[l];                           从底部2×2子式移位
        nm=k-1;
        y=w[nm];
        g=rv1[nm];
        h=rv1[k];
        f=((y-z)*(y+z)+(g-h)*(g+h))/(2.0*h*y);
        g=pythag(f,1.0);
        f=((x-z)*(x+z)+h*((y/(f+SIGN(g,f)))-h))/x;
        c=s=1.0;                          下面进行QR变换
        for (j=l;j<=nm;j++) {
            i=j+1;
            g=rv1[i];
            y=w[i];
            h=s*g;
            g=c*g;
            z=pythag(f,h);
            rv1[j]=z;
            c=f/z;
            s=h/z;
            f=x*c+g*s;
            g = g*c-x*s;
            h=y*s;
            y *= c;
            for (jj=1;jj<=n;jj++) {
                x=v[jj][j];
                z=v[jj][i];
                v[jj][j]=x*c+z*s;
                v[jj][i]=z*c-x*s;
            }
            z=pythag(f,h);
            w[j]=z;                       如果z=0，可循环任意次。
            if (z) {
                z=1.0/z;
                c=f*z;
                s=h*z;
            }
            f=c*g+s*y;
            x=c*y-s*g;

            for (jj=1;jj<=m;jj++) {
                y=a[jj][j];
                z=a[jj][i];
                a[jj][j]=y*c+z*s;
                a[jj][i]=z*c-y*s;
            }
        }
        rv1[l]=0.0;
        rv1[k]=f;
        w[k]=x;
    }
  }
  free_vector(rv1,1,n);
}
```

```
#include <math.h>
#include "nrutil.h"

float pythag(float a, float b)  计算(a²+b²)^1/2,没有破坏性的上溢和下溢.
{
    float absa,absb;
    absa=fabs(a);
    absb=fabs(b);
    if(absa>absb) return absa * sqrt(1.0--SQR(absb/absa));
    else return (absb==0.0 ? 0.0: absb * sqrt(1.0+SQR(absa/absb)));
}
```

(**svdcmp**,**svbksb** 和 **pythag** 程序的双精度版本,我们称为 **dsvdcmp**,**dsvbksb** 和 **dpythag**,它们被用于第5.13节中程序 **ratlsq**。读者可以很容易实现数据转换,或者从数值算法程序磁盘中获得这些转换好的程序。)

参考文献和进一步读物:


Forsythe, G. E., Malcolm, M. A., and Moler, C. B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 9. [1]

Wilkinson, J. H., and Reinsch, C. 1971, *Linear Algedra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag), Chapter I. 10 by G. H. Golub and G. Reinsch. [2]

Dongarra, J. J., et al. 1979, *LINPACK User's Guide* (Philadelphia: S. I. A. M.), Chapter 11. [3]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §6.7. [4]

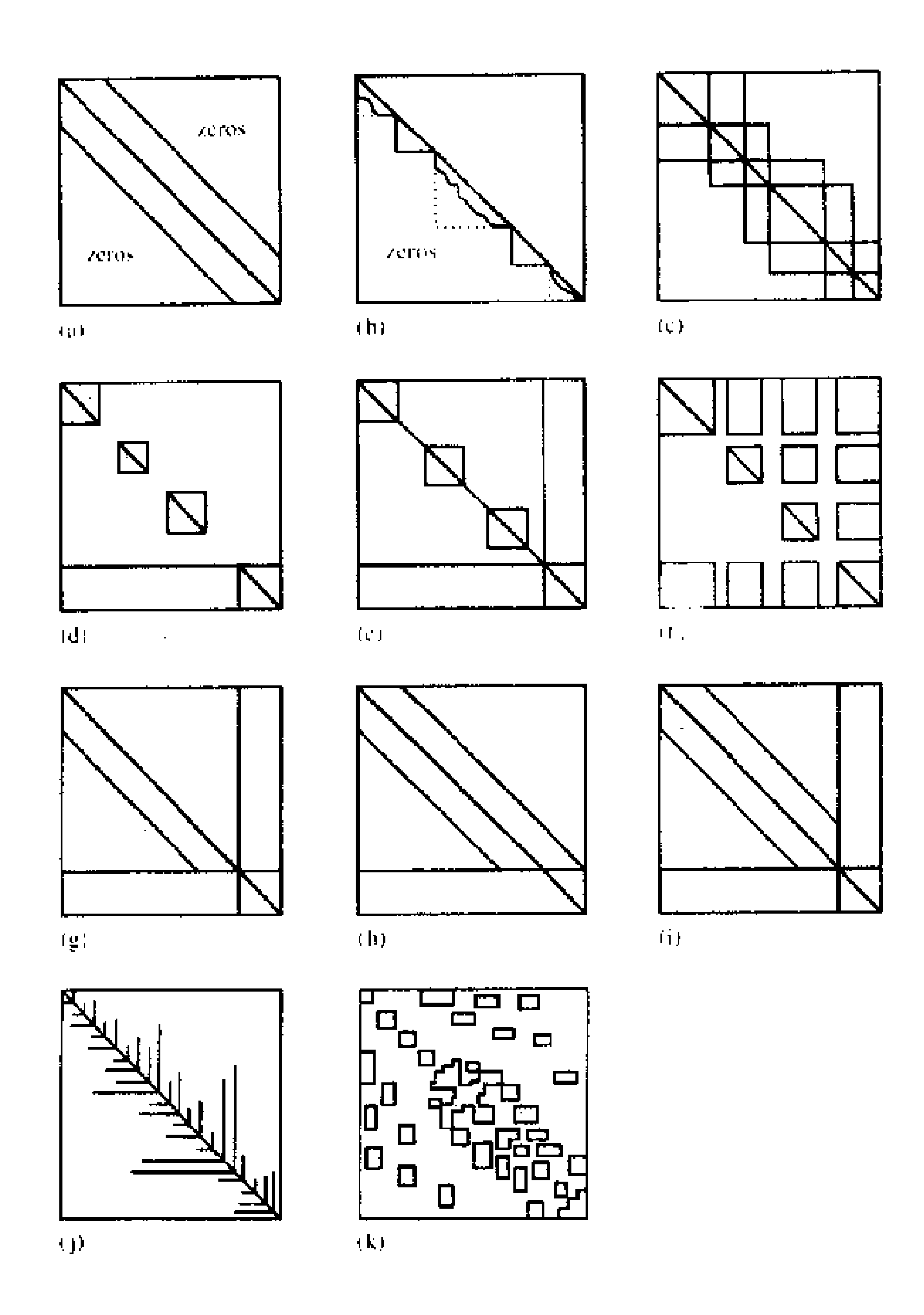
Golub, G. H., and Van Loan, C. F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §5.2.6. [5]


## 2.7 稀疏线性系统

如果一个线性方程组系统,仅有相对很少的一部分元素 $a_{ij}$ 为非零,则称其为**稀疏**的。对这类问题,用通用的线性系统求解方法是浪费的,因为在求解或求逆矩阵时,运算量为 $O(N^3)$ 的大部分算术运算都用来处理零元素。更有甚者,希望求解的问题可能大到已超出了可用的存储空间,而保留这些无用的零却很浪费空间。注意,对任意稀疏矩阵的求解方法,不外乎有两个明确的目的,(它们并不总能同时达到):省时间并且(或者)省空间。

在第2.4节中,我们已考虑过一种原始的稀疏矩阵模型,带状对角矩阵。在那里我们看到,可能做到既省时间($N^3$降至 $N$ 数量级),又省空间($N^2$降至 $N$ 数量级)。求解的算法原则上无异于一般的 $LU$ 分解法,不过是注意到如何记录零元素而应用得巧妙些罢了。大部分求解稀疏矩阵的实际问题都有同样的特征。它们基本上都采用分解法或者类似高斯——若当的消去法,但必须进行认真的优化,以减少被称为填充项的数目,这些填充项最初是零元素,而在求解过程中必然变成非零元素,为此必须保留它们的存储空间。

求解稀疏方程组的直接方法则主要是依赖于矩阵的稀疏模式。常见的模式,或者在化简一般形式矩阵时所用的一些有用子模式,都已有了特定的名称及特定的解法。这里不再费篇幅详细介绍了,参考书目列在本节最后,读者可以自学一些更专业的内容。下面列出的各词(及图2.7.1)可供使用。




($a$)带状对角矩阵；($b$)块状三角矩阵；($c$)块状三角矩阵；($d$)单边镶边块状对角矩阵；($e$)双边镶边块状对角矩阵；($f$)单边镶边块状三角矩阵；($g$)镶边带状三角矩阵；($h$)和($i$)分别为单边和双边镶边带状对角矩阵；($j$)和($k$)为其它形式。(见 Tewarson 的书[1])

图2.7.1　稀疏矩阵的一些标准形式

- 三对角矩阵
- 带宽为 $M$ 的带状对角矩阵(或带状矩阵)

- 带状三角矩阵
- 块状对角矩阵
- 块状三对角矩阵
- 块状三角矩阵
- 环带状矩阵
- 单边(或双边)镶边块状对角矩阵
- 单边(或双边)镶边块状三角矩阵
- 单边(或双边)镶边带状对角矩阵
- 单边(或双边)镶边带状三角矩阵
- 其它

此外,还应知道一些在求解二维或高维偏微分方程时,遇到的特殊的稀疏矩阵形式,参见第十九章。

如果特殊的稀疏模式不是单一的,那么读者可能希望试用**分析/因子分解/运算程序包**,它能自动地计算出怎样减少填充项。分析步骤对每个稀疏模式只做一次。因子分解步骤对每个适合该模式的特殊矩阵做一次。运算步骤对每个使用该特殊矩阵的右端项进行一次。有关的参考文献参见 Jacobs 的著作[2,3]。NAG 库[4]具有分析/因子分解/运算的能力。从 IMSL[5]那儿也可得到一个稀疏矩阵计算的程序集,如同耶鲁大学稀疏矩阵程序包[6]。

应当清楚,跟类似用主元消去的正规 $LU$ 分解法相比,因稀疏矩阵解法为减少填充项和算术运算次数,进行了特殊的次序交换和消去,一般来说,其算法的数值稳定性要有所降低。将求解的线性问题按比例变换其系数,使其非零元素之间可进行量值比较,这样(如果能做到的话)就会改善这种稳定性问题。

在本节剩下的内容中,我们提供两种方法,它们可用来求解一般的稀疏矩阵问题,而并不依赖于稀疏模式的具体情况。

### 2.7.1 谢尔曼-莫里森公式

假定读者已费力求出了方阵 $\mathbf{A}$ 的逆 $\mathbf{A}^{-1}$。现在,又想对 $\mathbf{A}$ 做点"小"变化,比如更换一个元素 $a_{ij}$,或是更换几个元素,或更换一行,或更换一列。有没有办法直接计算 $\mathbf{A}^{-1}$ 中相应有变化的项,而不必重复整个复杂的过程呢?有的,如果对向量 $\mathbf{u}$ 和 $\mathbf{v}$ 来说变换形式为

$$\mathbf{A} \to (\mathbf{A} + \mathbf{u} \otimes \mathbf{v}) \tag{2.7.1}$$

如果 $\mathbf{u}$ 为单位向量 $\mathbf{e}_i$,则式(2.7.1)将 $\mathbf{u}$ 的元素加到第 $i$ 行上(回顾一下,$\mathbf{u} \otimes \mathbf{v}$ 表示矩阵的第 $i$ 行、第 $j$ 列元素值为 $\mathbf{u}$ 的第 $i$ 项与 $\mathbf{v}$ 的第 $j$ 项之积)。如果 $\mathbf{v}$ 是单位向量 $\mathbf{e}_j$,则(2.7.1)将 $\mathbf{u}$ 的元素加到第 $j$ 列上。如果 $\mathbf{u}$ 和 $\mathbf{v}$ 恰好都是单位向量 $\mathbf{e}_i$ 和 $\mathbf{e}_j$,则相应的项仅加在元素 $a_{ij}$ 上。

谢尔曼-莫里森公式给出逆矩阵 $(\mathbf{A}+\mathbf{u}\otimes\mathbf{v})^{-1}$,推导如下:

$$\begin{aligned}
(\mathbf{A} + \mathbf{u} \otimes \mathbf{v})^{-1} &= (\mathbf{1} + \mathbf{A}^{-1} \cdot \mathbf{u} \otimes \mathbf{v})^{-1} \cdot \mathbf{A}^{-1} \\
&= (\mathbf{1} - \mathbf{A}^{-1} \cdot \mathbf{u} \otimes \mathbf{v} + \mathbf{A}^{-1} \cdot \mathbf{u} \otimes \mathbf{v} \cdot \mathbf{A}^{-1} \cdot \mathbf{u} \otimes \mathbf{v} - \cdots) \cdot \mathbf{A}^{-1} \\
&= \mathbf{A}^{-1} - \mathbf{A}^{-1} \cdot \mathbf{u} \otimes \mathbf{v} \cdot \mathbf{A}^{-1}(1 - \lambda + \lambda^2 - \cdots) \\
&= \mathbf{A}^{-1} - \frac{(\mathbf{A}^{-1} \cdot \mathbf{u}) \otimes (\mathbf{v} \cdot \mathbf{A}^{-1})}{1 + \lambda}
\end{aligned} \tag{2.7.2}$$

其中

$$\lambda \equiv \mathbf{v} \cdot \mathbf{A}^{-1} \cdot \mathbf{u} \tag{2.7.3}$$

式(2.7.2)第二行为标准幂级数展开。第三行是内积和外积相结合将标量λ分解出来。

式(2.7.2)的用途在于：给定$\mathbf{A}^{-1}$及向量$\mathbf{u}$与$\mathbf{v}$，我们只需求出两个矩阵的乘积和一个向量的点积，

$$\mathbf{z} \equiv \mathbf{A}^{-1} \cdot \mathbf{u} \qquad \mathbf{w} \equiv (\mathbf{A}^{-1})^T \cdot \mathbf{v} \qquad \lambda = \mathbf{v} \cdot \mathbf{z} \tag{2.7.4}$$

便可得到欲求的变化后的逆

$$\mathbf{A}^{-1} \to \mathbf{A}^{-1} - \frac{\mathbf{z} \otimes \mathbf{w}}{1 + \lambda} \tag{2.7.5}$$

整个过程仅需要$3N^2$次乘法和次数差不多的加法(如果$\mathbf{u}$或$\mathbf{v}$为单位向量，则次数更少)。

谢尔曼-莫里森公式可直接用来求解一类稀疏矩阵问题。如果已有一个快速计算$\mathbf{A}$的逆的算法(例如三对角阵或其它标准稀疏矩阵的形式)，则式(2.7.4)～式(2.7.5)允许构造相关的但更为复杂形式的矩阵，例如每次添加一行或一列。注意，谢尔曼-莫里森公式可以连续使用一次以上，每次都用最近更新过的$\mathbf{A}^{-1}$(方程2.7.5)。自然，如果不得不修改每一行，则又回到了$N^3$次的方法了。$N^3$前面的常数虽然只比直接法大几倍，但失去了直接法稳定的长处——因而要谨慎使用。

对另外一些稀疏矩阵问题，谢尔曼-莫里森公式不能直接应用，原因很简单，要存储整个逆矩阵$\mathbf{A}^{-1}$是行不通的。如果仅要修改$\mathbf{u}\otimes\mathbf{v}$的一处，且求解的线性问题为

$$(\mathbf{A} + \mathbf{u} \otimes \mathbf{v}) \cdot \mathbf{x} = \mathbf{b} \tag{2.7.6}$$

则可按下面方法来做。对矩阵$\mathbf{A}$应用假定是通用的快速算法，来求解两个辅助问题

$$\mathbf{A} \cdot \mathbf{y} = \mathbf{b} \qquad \mathbf{A} \cdot \mathbf{z} = \mathbf{u} \tag{2.7.7}$$

求向量$\mathbf{y}$和$\mathbf{z}$，由此，将(2.7.2)式右边乘以$\mathbf{b}$，得

$$\mathbf{x} = \mathbf{y} - \left[\frac{\mathbf{v} \cdot \mathbf{y}}{1 + (\mathbf{v} \cdot \mathbf{z})}\right] \mathbf{z} \tag{2.7.8}$$

### 2.7.2 周期三对角系统

所谓的周期三对角系统会经常遇到，而且是用刚才所述的方法利用谢尔曼-莫里森公式求解的一个很好的示例。方程形式为

$$\begin{bmatrix} b_1 & c_1 & 0 & \cdots & & & \beta \\ a_2 & b_2 & c_2 & \cdots & & & \\ & & & \cdots & & & \\ & & & \cdots & a_{N-1} & b_{N-1} & c_{N-1} \\ \alpha & & & \cdots & 0 & a_N & b_N \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_{N-1} \\ x_N \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ \cdots \\ r_{N-1} \\ r_N \end{bmatrix} \tag{2.7.9}$$

这是一个三对角系统，只是角上有矩阵元素$\alpha$和$\beta$。这种形式典型地由具有周期性边界条件的有限差分微分方程产生(第19.4节)。

我们使用谢尔曼-莫里森公式，将该系统作为三对角系统加上一个修正处理。用方程(2.7.6)中的记法，将向量$\mathbf{u}$和$\mathbf{v}$定义为

$$\mathbf{u}=\begin{bmatrix}\gamma\\0\\\vdots\\\vdots\\0\\\alpha\end{bmatrix}\qquad \mathbf{v}=\begin{bmatrix}1\\0\\\vdots\\\vdots\\0\\\beta/\gamma\end{bmatrix}\tag{2.7.10}$$

此处 $\gamma$ 暂且是任意的。矩阵 **A** 是式(2.7.9)中的三对角部分，但有两个项修改过：

$$b'_1 = b_1 - \gamma\ ,\qquad b'_N = b_N - \alpha\beta/\gamma\tag{2.7.11}$$

现在，我们用标准的三对角算法来求解方程(2.7.7)，然后由等式(2.7.8)得到解。

下面的程序 **cyclic** 实现了该算法，我们取任意参数 $\gamma = -b_1$，以避免在(2.7.11)的第一个方程中造成精度损失。而在第二个方程中不会产生精度损失。因而可以做不同的选择。

```
#include "nrutil.h"

void cyclic (float a[],float b[], float c[],float alpha,float beta,float r[],float x[],unsigned long n)
    求方程(2.7.9)给出的“周期”线性方程组的一个解向量 x[1..n]。a,b,c 和 r 为输入向量，其维数均是[1..n]。alpha
    和 beta 是矩阵角上的元素值。输入量不被修改
{
    void tridag(float a[], float b[], float c[], float r[], float u[],
        unsigned long n);
    unsigned long i;
    float fact,gamma,*bb,*u,*z;

    if (n <= 2) nrerror("n too small in cyclic");
    bb=vector(1,n);
    u=vector(1,n);
    z=vector(1,n);
    gamma = -b[1];                                  形成 bb[]时避免出现减法误差
    bb[1]=b[1]-gamma;                               为修改过的三对角系统设立对角线
    bb[n]=b[n]-alpha*beta/gamma;
    for (i=2;i<n;i++) bb[i]=b[i];
    tridag(a,bb,c,r,x,n);                           求解 A·x = r
    u[1]=gamma;                                     设置向量 u
    u[n]=alpha;
    for (i=2;i<n;i++) u[i]=0.0;
    tridag(a,bb,c,u,z,n);                           求解 A·z = u
    fact=(x[1]+beta*x[n]/gamma)/                    形成 v·x/(1 + v·z).
        (1.0+z[1]+beta*z[n]/gamma);
    for (i=1;i<=n;i++) x[i] -= fact*z[i];           至此得到解向量 x
    free_vector(z,1,n);
    free_vector(u,1,n);
    free_vector(bb,1,n);
}
```

## 2.7.3 伍德伯瑞(Woodbury)公式

如果要修改超过一个的项，则不能重复地使用(2.7.8)式，因为没有存储新的 $\mathbf{A}^{-1}$，就不能在做完一次后高效地求解辅助问题(2.7.7)。取而代之，**伍德伯瑞公式**是谢尔曼—莫里森公式的成块矩阵转换形式。

$$(\mathbf{A}+\mathbf{U}\cdot\mathbf{V}^T)^{-1} = \mathbf{A}^{-1} - [\mathbf{A}^{-1}\cdot\mathbf{U}\cdot(\mathbf{1}+\mathbf{V}^T\cdot\mathbf{A}^{-1}\cdot\mathbf{U})^{-1}\cdot\mathbf{V}^T\cdot\mathbf{A}^{-1}]\tag{2.7.12}$$

其中 **A** 跟往常一样是 $N\times N$ 矩阵，而 **U** 和 **V** 是 $N\times P$ 矩阵，$P<N$，且通常是 $P\ll N$，如果下图表示要修改项的中间部分，可能会更清楚些。

$$\left[\ \mathbf{U}\ \right]\cdot\left[\ -\mathbf{V}^T\cdot\mathbf{A}^{-1}\cdot\mathbf{U}\ \right]\cdot\left[\ \mathbf{V}^T\ \right]\tag{2.7.13}$$

可以看出，其中要求逆的矩阵是 $P\times P$，而不是 $N\times N$。

如注意到下面的事实，伍德伯瑞公式与连续使用谢尔曼—莫里森公式之间的联系就很清楚了。如果矩阵 $\mathbf{U}$ 由 $P$ 个向量 $\mathbf{u}_1,\cdots,\mathbf{u}_P$ 按列构成，矩阵 $\mathbf{V}$ 由 $P$ 个向量 $\mathbf{v}_1,\cdots,\mathbf{v}_P$ 按列构成，

$$\mathbf{U}\equiv\left[\mathbf{u}_1\ \cdots\ \mathbf{u}_P\right]\qquad\mathbf{V}\equiv\left[\mathbf{v}_1\ \cdots\ \mathbf{v}_P\right]\tag{2.7.14}$$

则对 $\mathbf{A}$ 的修改可用两种方法表示

$$\left(\mathbf{A}+\sum_{k=1}^{P}\mathbf{u}_k\otimes\mathbf{v}_k\right)=(\mathbf{A}+\mathbf{U}\cdot\mathbf{V}^T)\tag{2.7.15}$$

（注意 $\mathbf{u}$ 和 $\mathbf{v}$ 的下标并**不**表示元素，而是用来区分不同的列向量。）

方程(2.7.15)表明，如果 $\mathbf{A}^{-1}$ 已在存储器中，则可以用(2.7.12)求一个 $P\times P$ 矩阵的逆，便可一下子求出进行 $P$ 次修改后的结果，或者也可用(2.7.5)连续调用 $P$ 次得到。

如果没有现成存好的 $\mathbf{A}^{-1}$，则必须按下列方法使用式(2.7.12)求解：欲求解的线性方程为

$$\left(A+\sum_{k=1}^{P}\mathbf{u}_k\otimes\mathbf{v}_k\right)\cdot\mathbf{x}=\mathbf{b}\tag{2.7.16}$$

首先求解 $D$ 个辅助方程组

$$\begin{aligned}\mathbf{A}\cdot\mathbf{z}_1&=\mathbf{u}_1\\\mathbf{A}\cdot\mathbf{z}_2&=\mathbf{u}_2\\&\cdots\\\mathbf{A}\cdot\mathbf{z}_P&=\mathbf{u}_P\end{aligned}\tag{2.7.17}$$

并根据得到的 $\mathbf{z}$ 按列构造矩阵 $\mathbf{Z}$

$$\mathbf{Z}\equiv\left[\mathbf{z}_1\ \cdots\ \mathbf{z}_P\right]\tag{2.7.18}$$

第二步，求 $P\times P$ 矩阵的逆

$$\mathbf{H}\equiv(1+\mathbf{V}^T\cdot\mathbf{Z})^{-1}\tag{2.7.19}$$

最后，再求解一个辅助方程组

$$\mathbf{A}\cdot\mathbf{y}=\mathbf{b}\tag{2.7.20}$$

根据已有这些量，则可解得

$$\mathbf{x}=\mathbf{y}-\mathbf{Z}\cdot\left[\mathbf{H}\cdot(\mathbf{V}^T\cdot\mathbf{Y})\right]\tag{2.7.21}$$

## 2.7.4 分区求逆

有时候会遇到一个矩阵（尽管不必是稀疏的），它可以通过分区来高效地求逆。假定 $N\times N$ 矩阵 $\mathbf{A}$ 被划分成

$$\mathbf{A}=\begin{bmatrix}\mathbf{P}&\mathbf{Q}\\\mathbf{R}&\mathbf{S}\end{bmatrix}\tag{2.7.22}$$

其中 **P** 和 **S** 相应为 $p\times p$ 和 $s\times s$ 方阵($p+s=N$)。矩阵 **Q** 和 **R** 不必是方阵，相应为 $p\times s$ 和 $s\times p$ 的矩阵。

如果 **A** 的逆被划为同样的形式，

$$\mathbf{A}^{-1}=\begin{bmatrix}\widetilde{\mathbf{P}} & \widetilde{\mathbf{Q}}\\ \widetilde{\mathbf{R}} & \widetilde{\mathbf{S}}\end{bmatrix} \tag{2.7.23}$$

则 $\widetilde{\mathbf{P}},\widetilde{\mathbf{Q}},\widetilde{\mathbf{R}},\widetilde{\mathbf{S}}$，与相应的 **P**,**Q**,**R**,**S** 大小相同，可以由下面的公式求出：

$$\begin{aligned}\widetilde{\mathbf{P}}&=(\mathbf{P}-\mathbf{Q}\cdot\mathbf{S}^{-1}\cdot\mathbf{R})^{-1}\\ \widetilde{\mathbf{Q}}&=-(\mathbf{P}-\mathbf{Q}\cdot\mathbf{S}^{-1}\cdot\mathbf{R})^{-1}\cdot(\mathbf{Q}\cdot\mathbf{S}^{-1})\\ \widetilde{\mathbf{R}}&=-(\mathbf{S}^{-1}\cdot\mathbf{R})\cdot(\mathbf{P}-\mathbf{Q}\cdot\mathbf{S}^{-1}\cdot\mathbf{R})^{-1}\\ \widetilde{\mathbf{S}}&=\mathbf{S}^{-1}+(\mathbf{S}^{-1}\cdot\mathbf{R})\cdot(\mathbf{P}-\mathbf{Q}\cdot\mathbf{S}^{-1}\cdot\mathbf{R})^{-1}\cdot(\mathbf{Q}\cdot\mathbf{S}^{-1})\end{aligned} \tag{2.7.24}$$

或由其等价的公式求出：

$$\begin{aligned}\widetilde{\mathbf{P}}&=\mathbf{P}^{-1}+(\mathbf{P}^{-1}\cdot\mathbf{Q})\cdot(\mathbf{S}-\mathbf{R}\cdot\mathbf{P}^{-1}\cdot\mathbf{Q})^{-1}\cdot(\mathbf{R}\cdot\mathbf{P}^{-1})\\ \widetilde{\mathbf{Q}}&=-(\mathbf{P}^{-1}\cdot\mathbf{Q})\cdot(\mathbf{S}-\mathbf{R}\cdot\mathbf{P}^{-1}\cdot\mathbf{Q})^{-1}\\ \widetilde{\mathbf{R}}&=-(\mathbf{S}-\mathbf{R}\cdot\mathbf{P}^{-1}\cdot\mathbf{Q})^{-1}\cdot(\mathbf{R}\cdot\mathbf{P}^{-1})\\ \widetilde{\mathbf{S}}&=(\mathbf{S}-\mathbf{R}\cdot\mathbf{P}^{-1}\cdot\mathbf{Q})^{-1}\end{aligned} \tag{2.7.25}$$

等式(2.7.24)和(2.7.25)中的小括号指出了只需计算一次重复因子(当然，根据结合律，矩阵相乘的次序随便)。选择使用等式(2.7.24)或是等式(2.7.25)，取决于或是要求 $\widetilde{\mathbf{P}}$ 或 $\widetilde{\mathbf{S}}$ 有简单的公式；或是重复的表达式$(\mathbf{S}-\mathbf{R}\cdot\mathbf{P}^{-1}\cdot\mathbf{Q})^{-1}$要比表达式$(\mathbf{P}-\mathbf{Q}\cdot\mathbf{S}^{-1}\cdot\mathbf{R})^{-1}$计算简单；或是 **P** 和 **S** 的相对大小；或是是否 $\mathbf{P}^{-1}$或 $\mathbf{S}^{-1}$已知。

另一个有时有用的公式是，求分区矩阵的行列式，

$$\begin{aligned}\det\mathbf{A}&=\det\mathbf{P}\,\det(\mathbf{S}-\mathbf{R}\cdot\mathbf{P}^{-1}\cdot\mathbf{Q})\\ &=\det\mathbf{S}\,\det(\mathbf{P}-\mathbf{Q}\cdot\mathbf{S}^{-1}\cdot\mathbf{R})\end{aligned} \tag{2.7.26}$$

## 2.7.5 稀疏矩阵的索引存储

在第2.4节中，我们已经看到，三对角或带状对角矩阵可以用压缩格式存储，仅为那些可能不为零的元素分配存储空间，以及为保存方便可能会浪费掉几个存储单元。对更为一般的稀疏矩阵情况如何呢？对一个维数为 $N\times N$ 的稀疏矩阵，当它仅包含几倍于 $N$ 的非零元(一种典型情况)时，当然是低效的了，而且经常是，要为所有 $N^2$ 个元素分配空间，在物理上是不可能的，即使可以分配这么大的空间，它也是低效的或应当禁止的，因为它把大量机器时间用于循环查找这些非零元了。

显然需要一些索引存储方案，它仅保存非零的矩阵元素，而且具有足够的辅助信息，以确定一个元素的逻辑位置，及如何使各个元素对某共同的矩阵运算能循环操作起来。不幸的是，并不存在一个通用的标准方案。Knuth[7]中描述了一种方法。Yale Sparse Matrix Package[6]和 ITPACK[8]中描述了另个几种方法。对大多数应用，我们喜欢用 PCGPACK[9]中的存储方案，它跟 Bentley[10]中描述的方法差不多相同，而且与 Yale Sparse Matrix Package 中的一种也很相似。这种可以称作**行索引稀疏存储方式**的方案，仅需大约两倍于非零矩阵元素数目的存储量(其它方法可能需要三至五倍)。为简便起见，我们仅讨论实践中经常出现的方阵的情况。

为表示一个 $N\times N$ 维的矩阵 **A**，行索引方案设了两个一维的数组，称作 sa 和 ija。第一个用所需的单精度或双精度数存储矩阵元素的值，第二个存整数值。存储规则为：

- sa 的前 $N$ 个位置按顺序存储 **A** 的对角线矩阵元素。(注意，对角线元素即使为零也要存，因为实际应用中大部分情况对角线元素非零，故最多造成一点轻微的存储浪费)。

• ija 的前 $N$ 个位置保存数组 sa 的索引号，它指向矩阵相应行的第一个不在对角线上的元素（如果那一行没有不在对角线上的元素，则其内容为上一行中最后存储元素在 sa 中的索引加1）。

• ija 的第一个位置总是等于 $N+2$。（可据此判定 $N$）。

• ija 的第 $N+1$个位置是最后一行最后一个非对角元在 sa 中的索引号加1，（可据此判定矩阵非零元的数目，或数组 sa 和 ija 的元素个数）。sa 的第 $N+1$个位置不用，其值任意。

• sa 中$\geqslant N+2$的位置存有 **A** 不在对角线上的元素的值，它们按行排列，每行内按列排列。

• ija 中$\geqslant N+2$的位置保存的是 sa 中对应索引号中元素的列值。

尽管这些规则第一眼看上去很任意，但实际上它们是一个非常精巧的存储方案。举例说明，考虑矩阵

$$
\begin{bmatrix}
3. & 0. & 1. & 0. & 0. \\
0. & 4. & 0. & 0. & 0. \\
0. & 7. & 5. & 9. & 0. \\
0. & 0. & 0. & 0. & 2. \\
0. & 0. & 0. & 6. & 5.
\end{bmatrix}
\tag{2.7.27}
$$

用行索引压缩存储，矩阵(2.7.27)用两个长度为11的数组来表示，见表2.7.5.1。

**表 2.7.5.1**

| 索引号 k | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ija[k] | 7 | 8 | 8 | 10 | 11 | 12 | 3 | 2 | 4 | 5 | 4 |
| sa[k] | 3. | 4. | 5. | 0. | 5. | $x$ | 1. | 7. | 9. | 2. | 6. |

这里 $x$ 的值是任意的。注意，根据存储规则可知，$N$ 的值(即5)为 ija[1]−2，每个数组的长度为 ija[ija[1]−1]−1，即11。行 i 的对角线元素为 sa[i]，该行非对角线元素在 sa[k]中，其中 k 从 ija[i]到 ija[i+1]−1，当然上限要大于或等于下限(如在 C 语言中的循环)。

这里有一个程序 **sprsin**，它把一个矩阵从全部存储的方式转换成行索引稀疏存储方式，它把小于指定阈值的元素都视为零元抛弃了。当然，稀疏存储方式的主要用途是对那些用全部存储方式机器根本存不下的矩阵，那时就不得不直接产生稀疏格式。然而，**sprsin** 作为存储方案的一个精确的算法定义还是有用的，可用作大问题的子规模测试，以及对执行时间大于存储的场合，也迫使采用稀疏存储。

```
#include <math.h>

void sprsin(float **a,int n,float thresh,unsigned long nmax,float sa[], unsigned long ija[])
  将方阵a[1..n][1..n]转换成行索引稀疏存储方式。仅保留数值大于阈值的元素。输出两个维数为 nmax(作为输入
  参数)的线性数组，sa[1..n]包含数组值，ija[1..n]为索引号。填入输出 sa 和 ija 中的元素的数目都是 ija[ija[1]-1]
  -1。
{
    void nrerror(char error_text[]);
    int i,j;
    unsigned long k;

    for (j=1;j<=n;j++) sa[j]=a[j][j];              存对角线元
    ija[1]=n+2;                                    如果存，指向第一行的非对角元
    k=n+1;
    for (i=1;i<=n;i++) {                           行循环
        for (j=1;j<=n;j++) {                       列循环
            if (fabs(a[i][j]) >= thresh && i != j) {
                if (++k > nmax) nrerror("sprsin: nmax too small");
                sa[k]=a[i][j];                     保存非对角元及其列值
```

```
			ija[k]=j;
		}
	ija[i+1]=k+1;                      每行完成时，将指针指向下一个
	}
}
```

用行索引稀疏存储方式保存的矩阵的唯一最重要的用途就是在其右边乘以一个向量。事实上，存储方式的优化就是为了这个目的，因而下面的程序就非常简单了。

```
void sprsax(float sa[],unsigned long ija[],float x[],float b[],unsigned long n)
  将行索引稀疏存储数组 sa 和 ija 乘以一个向量 x[1..n]，得到向量 b[1..n]。
{
	void nrerror(char error_text[]);
	unsigned long i,k;

	if (ija[1] != n+2) nrerror("sprsax: mismatched vector and matrix");
	for (i=1;i<=n;i++) {
		b[i]=sa[i]*x[i];                           从对角项开始
		for (k=ija[i];k<=ija[i+1]-1;k++)           对非对角项循环
			b[i] += sa[k]*x[ija[k]];
		}
}
```

要是在矩阵的转置右边乘以一个向量也很简单。(本节后面，我们要用到这种运算。)注意，转置矩阵并未实际构造。

```
void sprstx(float sa[], unsigned long ija[], float x[], float b[], unsigned long n)
  将存储成数组 sa 和 ija 之行索引稀疏矩阵的转置乘以向量 x[1..n]，得到向量 b[1..n]。
{
	void nrerror(char error_text[]);
	unsigned long i,j,k;

	if (ija[1] != n+2) nrerror("mismatched vector and matrix in sprstx");
	for (i=1;i<=n;i++) b[i]=sa[i]*x[i];                 从对角项开始
	for (i=1;i<=n;i++) {                                 对非对角项循环
		for (k=ija[i];k<=ija[i+1]-1;k++) {
			j=ija[k];
			b[j] += sa[k]*x[i];
		}
	}
}
```

(双精度版的 **sprsax** 和 **sprstx** 称作 **dsprsax** 和 **dsprstx**，在本节后面的程序 **atimes** 中还要用到。读者可以很容易地进行转换，或是从本书磁盘中得到转换好的程序。)

事实上，因为选择行索引存储对行和列的处理完全不同，则常遇到要对一个给定的、以行索引稀疏存储方式存储的矩阵，构造其转置的操作。当该操作不可避免时，可以按下面的步骤来做：按列构造非对角元的索引(参见第8.4节)。则这些元素按列次序写入输出数组。在写每个元素时，要确定其行值并保存起来。最后，每列的元素按行次序存储。

```
void sprstp(float sa[], unsigned long ija[], float sb[], unsigned long ijb[])
  构造稀疏方阵的转置，从行索引稀疏存储的数组 sa 和 ija 转换成数组 sb 和 ijb。
{
	void iindexx(unsigned long n, long arr[], unsigned long indx[]);
	将程序indexx所有浮点型变量改成长型变量的版本
	unsigned long j,jl,jm,jp,ju,k,m,n2,noff,inc,iv;
	float v;
```

```
    n2=ija[1];                                              矩阵线性长度加2
    for (j=1;j<=n2-2;j++) sb[j]=sa[j];                      对角元
    iindexx(ija[n2-1]-ija[1],(long *)&ija[n2-1],&ijb[n2-1]);
    按列构造非对角元的索引
    jp=0;
    for (k=ija[1];k<=ija[n2-1]-1;k++) {                     对非对角元循环
        m=ijb[k]+n2-1;                                      使用索引表按(以前的)列存储
        sb[k]=sa[m];
        for (j=jp+1;j<=ija[m];j++) ijb[j]=k;                 将索引填入省略的行中
        jp=ija[m];                                          用二分法查找元素m在哪一行,并将其存入
        jl=1;                                                 ijb[k]
        ju=n2-1;
        while (ju-jl > 1) {
            jm=(ju+jl)/2;
            if (ija[jm] > m) ju=jm; else jl=jm;
        }
        ijb[k]=jl;
    }
    for (j=jp+1;j<n2;j++) ijb[j]=ija[n2-1];
    for (j=1;j<=n2-2;j++) {                                 用Shell排序算法对每行进行最后一遍排序
        jl=ijb[j+1]-ijb[j];
        noff=ijb[j]-1;
        inc=1;
        do {
            inc *= 3;
            inc++;
        } while (inc <= jl);
        do {
            inc /= 3;
            for (k=noff+inc+1;k<=noff+jl;k++) {
                iv=ijb[k];
                v=sb[k];
                m=k;
                while (ijb[m-inc] > iv) {
                    ijb[m]=ijb[m-inc];
                    sb[m]=sb[m-inc];
                    m -= inc;
                    if (m-noff <= inc) break;
                }
                ijb[m]=iv;
                sb[m]=v;
            }
        } while (inc > 1);
    }
}
```

上面的程序中,包含了第8.1节中的一个排序算法,但调用了外部程序 **iindexx** 来构造初始的列索引.该程序与第8.4节中所列的 **indexx** 相同,只是后者的两个浮点数说明应改为长型(long)。事实上,读者可以经常使用 **indexx** 而勿需修改,因为许多计算机不管数值被解释成浮点的还是整型的值,都会正确地独立地排序。

作为稀疏矩阵运算的最后一个示例,我们给出两个稀疏矩阵相乘的两个程序。它们对第13.10节中所描述的技术是很有用的。

一般说来,两个稀疏矩阵的积本身不是稀疏的。因此要限制积矩阵的大小有两种方法:其一,是事先已知积为特定的稀疏模式,仅计算其中的这些元素;其二,计算所有的非零元,但仅保存那些数值在阈值以上的元素。第一种技术如果能够使用是非常有效的。稀疏模式可以通过用行索引稀疏存储格式设置的数组(如 ija)来说明。并且程序还构造一个相应的值数组(如 sa)。第二种技术有超出计算时间及不知输出规模的危险,故应谨慎使用。

对行索引存储，一种很自然的乘法运算是，左边的矩阵乘以右边矩阵的转置，这样可以进行行与行的运算，而不是行对列。因此我们的程序是计算 $\mathbf{A}\cdot\mathbf{B}^T$，而不是 $\mathbf{A}\cdot\mathbf{B}$。这意味着在把右端矩阵提交矩阵的乘程序之前，必须运行 **sprstp** 程序对其进行转置。

两个实现程序，**sprspm** 进行"模式乘"，而 **sprstm** 进行"阈值乘"，它们在结构上非常相象。由于为两个输入流和输出流进行了不同的对角线或非对角线元素组合，两个算法都很复杂。

```
void sprspm(float sa[], unsigned long ija[], float sb[], unsigned long ijb[], float sc[],
    unsigned long ijc[])
```

矩阵乘法运算 $\mathbf{A}\cdot\mathbf{B}^T$，其中 **A** 和 **B** 是以行索引存储方式存储的两个稀疏矩阵，$\mathbf{B}^T$ 为 **B** 的转置。这里 sa 和 ija 存储矩阵 **A**，sb 和 ijb 存储矩阵 **B**。本程序仅计算由输入索引数组 ijc 预说明的矩阵乘积中的元素，ijc 本身不被修改。输入数组 sc 和 ijc 一起给出了乘积矩阵的行索引存储方式。对稀疏矩阵乘法，执行本程序前，经常应先调用 sprstp，以构造已知矩阵的转置，并存入到数组 sb，ijb。

```
{
    void nrerror(char error_text[]);
    unsigned long i,ijma,ijmb,j,m,ma,mb,mbb,mn;
    float sum;

    if (ija[1] != ijb[1] || ija[1] != ijc[1])
        nrerror("sprspm: sizes do not match");
    for (i=1;i<=ijc[1]-2;i++) {                          // 行循环
        j=m=i;                                            // 设置初值，以便第一遍循环处理对角元
        mn=ijc[i];
        sum=sa[i]*sb[i];
        for (;;) {                                        // 对每个输出元进行的主循环
            mb=ijb[j];
            for (ma=ija[i];ma<=ija[i+1]-1;ma++) {         // 对 A 的行元素循环。下面的逻辑是对对角元
                                                          // 及非对称元进行各种组合
                ijma=ija[ma];
                if (ijma == j) sum += sa[ma]*sb[j];
                else {
                    while (mb < ijb[j+1]) {
                        ijmb=ijb[mb];
                        if (ijmb == i) {
                            sum += sa[i]*sb[mb++];
                            continue;
                        } else if (ijmb < ijma) {
                            mb++;
                            continue;
                        } else if (ijmb == ijma) {
                            sum += sa[ma]*sb[mb++];
                            continue;
                        }
                        break;
                    }
                }
            }
            for (mbb=mb;mbb<=ijb[j+1]-1;mb++) {          // 处理完 B 的行的剩余部分
                if (ijb[mbb] == i) sum += sa[i]*sb[mbb];
            }
            sc[m]=sum;
            sum=0.0;                                      // 为下一次循环重置索引
            if (mn >= ijc[i+1]) break;
            j=ijc[m=mn++];
        }
    }
}
```

```
#include <math.h>
```

```
void sprstm(float sa[], unsigned long ija[], float sb[], unsigned long ijb[],
	float thresh, unsigned long nmax, float sc[], unsigned long ijc[])
```

矩阵乘法运算 $\mathbf{A}\cdot\mathbf{B}^T$，其中 $\mathbf{A}$ 和 $\mathbf{B}$ 是以行索引存储方式存储的两个稀疏矩阵，$\mathbf{B}^T$ 为 $\mathbf{B}$ 的转置。这里 sa 和 ija 存储矩阵 $\mathbf{A}$，sb 和 ijb 存储矩阵 $\mathbf{B}$。本程序计算矩阵积中的所有元素（可能不是稀疏的！），但仅存储那些数值超过阈值的元素。输出数组 sc 和 ijc（其最大长度由输入量 nmax 确定）用行索引存储方式给出积矩阵。对稀疏矩阵乘法，执行本程序前应先调用 sprstp，以构造已知矩阵的转置，并存入到 sb 和 ijb。

```
	void nrerror(char error_text[]);
	unsigned long i,ijma,ijmb,j,k,ma,mb,mbb;
	float sum;

	if (ija[1] != ijb[1]) nrerror("sprstm: sizes do not match");
	ijc[1]=k=ija[1];
	for (i=1;i<=ija[1]-2;i++) {                      A 的行循环
		for (j=1;j<=ijb[1]-2;j++) {                  B 的行循环
			if (i == j) sum=sa[i]*sb[j]; else sum=0.0e0;
			mb=ijb[j];
			for (ma=ija[i];ma<=ija[i+1]-1;ma++) {    对 A 的行元素循环，下面复杂的逻辑是对
			                                         对角元及非对角元进行各种组合
				ijma=ija[ma];
				if (ijma == j) sum += sa[ma]*sb[j];
				else {
					while (mb < ijb[j+1]) {
						ijmb=ijb[mb];
						if (ijmb == i) {
							sum += sa[i]*sb[mb++];
							continue;
						} else if (ijmb < ijma) {
							mb++;
							continue;
						} else if (ijmb == ijma) {
							sum += sa[ma]*sb[mb++];
							continue;
						}
						break;
					}
				}
			}
			for (mbb=mb;mbb<=ijb[j+1]-1;mbb++) {     处理完 B 的行的剩余部分
				if (ijb[mbb] == i) sum += sa[i]*sb[mbb];
			}
			if (i == j) sc[i]=sum;                   输出结果
			else if (fabs(sum) > thresh) {
				if (k > nmax) nrerror("sprstm: nmax too small");
				sc[k]=sum;
				ijc[k++]=j;
			}
		}
		ijc[i+1]=k;
	}
}
```

## 2.7.6 共轭梯度法求解稀疏系统

所谓的**共轭梯度法**，为求解 $N\times N$ 线性系统提供了一种相当通用的方法：

$$\mathbf{A}\cdot\mathbf{x}=\mathbf{b} \tag{2.7.29}$$

这些求解大型稀疏系统的方法，其好处在于它们对 $\mathbf{A}$ 的引用仅包括 $\mathbf{A}$ 与一个向量相乘，或其转置与

个向量相乘。正如我们所见，这些运算对适当存储的稀疏矩阵是非常高效的。读者，可能会要求提供一些函数，希望它们对这些稀疏矩阵乘法尽可能高效地运行。而我们现在提供一些通用的程序，下面的 linbcg，它能求解线性方程组(2.7.29)。

最简单的常见共轭梯度算法只能求解(2.7.29)，是仅当 A 为对称的及正定的时候。它基于极小化下面函数的思想

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}\cdot\mathbf{A}\cdot\mathbf{x} - \mathbf{b}\cdot\mathbf{x} \tag{2.7.30}$$

当其梯度

$$\nabla f = \mathbf{A}\cdot\mathbf{x} - \mathbf{b} \tag{2.7.31}$$

为零时，该函数极小化，这相当于式(2.7.29)。极小化的实现是通过产生一个搜索方向 $\mathbf{p}_k$ 的后继以及改进极小化的因变量 $\mathbf{x}_k$。每一步都是要找一个量 $\alpha_k$ 来极小化 $f(\mathbf{x}_k+\alpha_k\mathbf{p}_k)$，并且把 $\mathbf{x}_{k+1}$ 设为等于新的点 $\mathbf{x}_k+\alpha_k\mathbf{p}_k$。$\mathbf{p}_k$ 和 $\mathbf{x}_k$ 也是这样构造的，$\mathbf{x}_{k+1}$ 是 $f$ 在所走过方向 $\{\mathbf{p}_1,\mathbf{p}_2,\cdots\mathbf{p}_k\}$ 上的向量空间内的极小点，经过 $N$ 次迭代后便可到达整个向量空间的极小点，即式(2.7.29)的解。

在后面的第10.6节中，我们将把这个"普通的"共轭梯度算法推广到任意非线性函数的情况。这里，我们的兴趣仍是求解线性的，但不必是正定的或对称的，其一种重要的推广是：**双共轭梯度法**。一般说来，该算法并非是简单的函数极小化问题。它要构造四个向量序列 $\mathbf{r}_k,\bar{\mathbf{r}}_k,\mathbf{p}_k,\bar{\mathbf{p}}_k,k=1,2\cdots$。只要提供初始向量 $\mathbf{r}_1$ 和 $\bar{\mathbf{r}}_1$，并设 $\mathbf{p}_1=\mathbf{r}_1,\bar{\mathbf{p}}_1=\bar{\mathbf{r}}_1$，则可得下面的递推式：

$$\begin{aligned}
\alpha_k &= \frac{\bar{\mathbf{r}}_k\cdot\mathbf{r}_k}{\bar{\mathbf{p}}_k\cdot\mathbf{A}\cdot\mathbf{p}_k}\\
\mathbf{r}_{k+1} &= \mathbf{r}_k - \alpha_k\mathbf{A}\cdot\mathbf{p}_k\\
\bar{\mathbf{r}}_{k+1} &= \bar{\mathbf{r}}_k - \alpha_k\mathbf{A}^T\cdot\bar{\mathbf{p}}_k\\
\beta_k &= \frac{\bar{\mathbf{r}}_{k+1}\cdot\mathbf{r}_{k+1}}{\bar{\mathbf{r}}_k\cdot\mathbf{r}_k}\\
\mathbf{p}_{k+1} &= \mathbf{r}_{k+1} + \beta_k\cdot\mathbf{p}_k\\
\bar{\mathbf{p}}_{k+1} &= \bar{\mathbf{r}}_{k+1} + \beta_k\cdot\bar{\mathbf{p}}_k
\end{aligned} \tag{2.7.32}$$

该向量序列满足**双正交**条件

$$\bar{\mathbf{r}}_i\cdot\mathbf{r}_j = \mathbf{r}_i\cdot\bar{\mathbf{r}}_j = 0,\quad j<i \tag{2.7.33}$$

及**双共轭条件**

$$\bar{\mathbf{p}}_i\cdot\mathbf{A}\cdot\mathbf{p}_j = \mathbf{p}_i\cdot\mathbf{A}^T\cdot\bar{\mathbf{p}}_j = 0,\ j<i \tag{2.7.34}$$

还是**相互正交的**

$$\bar{\mathbf{r}}_i\cdot\mathbf{p}_j = \mathbf{r}_i\cdot\bar{\mathbf{p}}_j = 0,\quad j<i \tag{2.7.35}$$

证明这些性质的直接推导见[14]。只要递推不因出现分母为零而提前中断，其结束必在 $m\leqslant N$ 步之内，且 $\mathbf{r}_{m+1}=\bar{\mathbf{r}}_{m+1}=0$。这是因为，最多 $N$ 步后就已知用完了所构造向量的新的正交方向。

使用该算法求解系统(2.7.29)，要先猜测一个初始的解 $\mathbf{x}_1$，将 $\mathbf{r}_1$ 选为**残差**

$$\mathbf{r}_1 = \mathbf{b} - \mathbf{A}\cdot\mathbf{x}_1 \tag{2.7.36}$$

并设 $\bar{\mathbf{r}}_1=\mathbf{r}_1$，然后构造改进的估算序列

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k\mathbf{p}_k \tag{2.7.37}$$

实现式(2.7.32)的递推。等式(2.7.37)保证递推得到的 $\mathbf{r}_{k+1}$，事实上是对应于 $\mathbf{x}_{k+1}$ 的残差 $\mathbf{b}-\mathbf{A}\cdot\mathbf{x}_{k+1}$。因为 $\mathbf{r}_{m+1}=0$，故 $\mathbf{x}_{m+1}$ 是方程(2.7.29)的解。

尽管不能保证整个过程不会中断，或是对一般的矩阵 A 会变得不稳定，然而实践中，这种情况很少见。更重要的是，仅当有精确的算法时，才会至多 $N$ 次迭代后便精确地结束。由于舍入误差，必须将运算看作一

个纯迭代的过程，当符合一些适当的错误准则时便应中止。

普通的共轭梯度算法是双共轭梯度算法的一个特例，即 $\mathbf{A}$ 是对称的，并选择 $\bar{\mathbf{r}}_1=\mathbf{r}_1$。然后对所有的 $k$，令 $\bar{\mathbf{r}}_k=\mathbf{r}_k$ 及 $\bar{\mathbf{p}}_k=\mathbf{p}_k$，可以省去这部分计算，将算法的工作量减半。共轭梯度法可解释为极小化等式(2.7.30)。如果 $\mathbf{A}$ 是正定及对称的，算法不会中断(理论上如此!)。下面的程序 linbcg 在输入对称的 A 时，实际上减化成普通的共轭梯度法了，但它却做了所有的冗余计算。

通用算法的另一种变化是对于对称的、但非正定的矩阵 $\mathbf{A}$，这时选择 $\bar{\mathbf{r}}_1=\mathbf{A}\cdot\mathbf{r}_1$ 而不是 $\bar{\mathbf{r}}_1=\mathbf{r}_1$。这种情况下，对所有的 $k$，$\bar{\mathbf{r}}_k=\mathbf{A}\cdot\mathbf{r}_k$ 及 $\bar{\mathbf{p}}_k=\mathbf{A}\cdot\mathbf{p}_k$。因此，算法等价于普通的共轭梯度算法，只是所有的点积运算 $\mathbf{a}\cdot\mathbf{b}$ 被 $\mathbf{a}\cdot\mathbf{A}\cdot\mathbf{b}$ 所代替。它被称作**极小残差算法**，因为它对应于下面函数的连续地极小化操作

$$\Phi(\mathbf{x}) = \frac{1}{2}\mathbf{r}\cdot\mathbf{r} = \frac{1}{2}|\mathbf{A}\cdot\mathbf{x} - \mathbf{b}|^2 \tag{2.7.38}$$

其中连续地迭代 $\mathbf{x}_k$，就可以在由共轭梯度法产生的搜索方向 $\mathbf{p}_k$ 相同的集上极小化 $\Phi$。该算法已推广为各种非对称矩阵的算法。**通用极小残差法**(GMRES；参见[9,15])可能是这些方法中最为稳健的了。

注意等式(2.7.38)给出

$$\nabla\Phi(\mathbf{x}) = \mathbf{A}^T\cdot(\mathbf{A}\cdot\mathbf{x} - \mathbf{b}) \tag{2.7.39}$$

对任意非奇异矩阵 $\mathbf{A}$，$\mathbf{A}^T\cdot\mathbf{A}$ 是对称的和正定的。因此，可以试着通过将普通的共轭梯度算法应用于下面问题，而来求解方程(2.7.29)

$$(\mathbf{A}^T\cdot\mathbf{A})\cdot\mathbf{x} = \mathbf{A}^T\cdot\mathbf{b} \tag{2.7.40}$$

但不要这样做!矩阵 $\mathbf{A}^T\cdot\mathbf{A}$ 的条件数是 $\mathbf{A}$ 的条件数的平方(条件数的定义见第2.6)。一个大的条件数不仅增加了所需的迭代次数，而且限制了可能得到的解的精度。用双共轭梯度法对原始矩阵 $\mathbf{A}$ 求解差不多总要好一些。

至此，我们尚未言及这些算法的收敛速度问题。对于条件好的矩阵，即“靠近”单位阵的矩阵，普通的共轭梯度算法效果很好。这提示我们把这些方法应用于方程(2.7.29)的先决条件形式，则

$$(\widetilde{\mathbf{A}}^{-1}\cdot\mathbf{A})\cdot\mathbf{x} = \widetilde{\mathbf{A}}^{-1}\cdot\mathbf{b} \tag{2.7.41}$$

这种想法是，对接近于 $\mathbf{A}$ 的某个 $\widetilde{\mathbf{A}}$ 求解线性系统可能要容易些，其中 $\widetilde{\mathbf{A}}^{-1}\cdot\mathbf{A}\approx\mathbf{1}$，它允许算法在较少的步数内收敛。矩阵 $\widetilde{\mathbf{A}}$ 称为一个先决条件[11]，这里给出的整个方案即为**有先决条件的双共轭梯度法**，或称为 PBCG。

为了能高效实现，PBCG 算法介绍了一个附加的向量集 $\mathbf{z}_k$ 和 $\bar{\mathbf{z}}_k$，定义为

$$\widetilde{\mathbf{A}}\cdot\mathbf{z}_k = \mathbf{r}_k \quad 及 \quad \widetilde{\mathbf{A}}^T\cdot\bar{\mathbf{z}}_k = \bar{\mathbf{r}}_k \tag{2.7.42}$$

并修改了等式(2.7.32)中 $\alpha_k$，$\beta_k$，$\mathbf{p}_k$ 及 $\bar{\mathbf{p}}_k$ 的定义

$$\begin{aligned}
\alpha_k &= \frac{\bar{\mathbf{r}}_k\cdot\mathbf{z}_k}{\bar{\mathbf{p}}_k\cdot\mathbf{A}\cdot\mathbf{p}_k} \\
\beta_k &= \frac{\bar{\mathbf{r}}_{k+1}\cdot\mathbf{z}_{k+1}}{\bar{\mathbf{r}}_k\cdot\mathbf{z}_k} \\
\mathbf{p}_{k+1} &= \mathbf{z}_{k+1} + \beta_k\mathbf{p}_k \\
\bar{\mathbf{p}}_{k+1} &= \bar{\mathbf{z}}_{k+1} + \beta_k\bar{\mathbf{p}}_k
\end{aligned} \tag{2.7.43}$$

对下面的程序 **linbcg**，我们要求读者自己来提供求解(2.7.42)的辅助线性系统的程序。如果不知道用什么作先决条件 $\widetilde{\mathbf{A}}$，则不妨使用 $\mathbf{A}$ 的对角线部分，甚至是单位矩阵，这样收敛的负担完全落在双共轭梯度算法本身上。

下面的 **linbcg** 程序，基于以前 Anne Greenbaum 所写的一个程序(参见[13]，它是一个不同的、稍简单些的实现)。还有一点技巧应该知道。

怎样能构成“好”的收敛是依赖于应用。因此程序 **linbcg** 提供四种选择，可以对输入标记 itol 进行设置。如果设 itol=1，则当值 $|\mathbf{A}\cdot\mathbf{x}-\mathbf{b}|/|\mathbf{b}|$ 小于输入量 tol 时，迭代停止。如果设 itol=2，则所需的准则为：

$$|\widetilde{\mathbf{A}}^{-1}\cdot(\mathbf{A}\cdot\mathbf{x}-\mathbf{b})|/|\widetilde{\mathbf{A}}^{-1}\cdot\mathbf{b}|<tol \qquad (2.7.[illegible])$$

如果 itol＝3，则程序使用其自身对 **x** 误差的估计，并要求其大小除以 x 的值小于 tol。设置 itol＝4 与 itol＝3 情况相同，只是用误差的最大量（绝对值）和 **X** 的最大分量代替了向量的大小（即，用 $L_\infty$ 标准代替了 $L_2$ 标准）。读者可能需要在实践中试着寻找到哪种收敛准则最适合自己的问题。

输出方面，err 是实际所达到的容差。如果返回的数 iter 表明所允许的最大迭代次数 itmax 未被超过，则 err 应比 tol 小。如果要进一步迭代，可不改变返回的量再次调用程序，然而两次调用之间，程序会丢失有关所跨越共轭梯度子空间的记忆，故不应使其经常返回，至少每次应有 N 次迭代。

最后，要注意的是，**linbcg** 中使用的是双精度数，因此它通常是在 N 相当大时使用的。

```
#include <stdio.h>
#include <math.h>
#include "nrutil.h"
#define EPS 1.0e-14

void linbcg(unsigned long n, double b[], double x[], int itol, double tol, int itmax, int *iter,
            double *err)
```

用迭代双共轭梯度法求解 A·x＝b，b[1..n]已知，求 x[1..n]。输入时 x[1..n]应设为解的初始猜测（或全为零）；itol 为1，2，3，或4，它说明使用何种收敛测试（见正文）；itmax 为所允许迭代的最大次数，tol 为所需的收敛容差。输出时 x[1..n]被置为改进后的解，iter 为实际进行的迭代次数，err 为估计的误差。矩阵 A 的引用仅通过用户提供的程序 **atimes**，它计算 A 或其转置与一个向量的乘积，而程序 **asolve**，它对某个先决条件矩阵 $\widetilde{A}$（可能仅是 A 的平凡对角线部分）求解 $\widetilde{A}\cdot x=b$ 或 $\widetilde{A}^T\cdot x=b$。

```
{
    void asolve(unsigned long n, double b[], double x[], int itrnsp);
    void atimes(unsigned long n, double x[], double r[], int itrnsp);
    double snrm(unsigned long n, double sx[], int itol);
    unsigned long j;
    double ak,akden,bk,bkden,bknum,bnrm,dxnrm,xnrm,zm1nrm,znrm;
    double *p,*pp,*r,*rr,*z,*zz;          本程序中采用双精度表示为好
    p=dvector(1,n);
    pp=dvector(1,n);
    r=dvector(1,n);
    rr=dvector(1,n);
    z=dvector(1,n);
    zz=dvector(1,n);

    计算初始残差，并检查停止准则
    *iter=0;
    atimes(n,x,r,0);                      atimes的输入为x[1..n]输出为r[1..n]，最后的0
    for (j=1;j<=n;j++) {                      表示使用矩阵本身(而不是其转置)
        r[j]=b[j]-r[j];
        rr[j]=r[j];
    }
/*  atimes(n,r,rr,0); */                  去掉本行的注释符号即得到该算法的"最小残差"
    znrm=1.0;                                 变种
    if (itol == 1) bnrm=snrm(n,b,itol);
    else if (itol == 2) {
        asolve(n,b,z,0);                  aslove的输入是b[1..n]，输出为z[1..n]；最后
        bnrm=snrm(n,z,itol);                  的0表示使用的是矩阵Ã[而不是其转置]
    }
    else if (itol == 3 || itol == 4) {
        asolve(n,b,z,0);
        bnrm=snrm(n,z,itol);
        asolve(n,r,z,0);
        znrm=snrm(n,z,itol);
    } else nrerror("illegal itol in linbcg");
    asolve(n,r,z,0);
    while (*iter <= itmax) {              主循环
        ++(*iter);
```

```
        zm1nrm=znrm;
        asolve(n,rr,zz,1);                    最后的 1 表示使用转置矩阵Ã^T
        for (bknum=0.0,j=1;j<=n;j++) bknum += z[j]*rr[j];
        计算导数bk及方向向量p和pp
        if (*iter == 1) {
            for (j=1;j<=n;j++) {
                p[j]=z[j];
                pp[j]=zz[j];
            }
        }
        else {
            bk=bknum/bkden;
            for (j=1;j<=n;j++) {
                p[j]=bk*p[j]+z[j];
                pp[j]=bk*pp[j]+zz[j];
            }
        }
        bkden=bknum;                          计算系数ak,新的迭代量x,及新的残差r和rr
        atimes(n,p,z,0);
        for (akden=0.0,j=1;j<=n;j++) akden += z[j]*pp[j];
        ak=bknum/akden;
        atimes(n,pp,zz,1);
        for (j=1;j<=n;j++) {
            x[j] += ak*p[j];
            r[j] -= ak*z[j];
            rr[j] -= ak*zz[j];
        }
        asolve(n,r,z,0);                      求解Ã·z=r,并检查停止准则
        if (itol == 1 || itol == 2) {
            znrm=1.0;
            *err=snrm(n,r,itol)/bnrm;
        } else if (itol == 3 || itol == 4) {
            znrm=snrm(n,z,itol);
            if (fabs(zm1nrm-znrm) > EPS*znrm) {
                dxnrm=fabs(ak)*snrm(n,p,itol);
                *err=znrm/fabs(zm1nrm-znrm)*dxnrm;
            } else {
                *err=znrm/bnrm;              误差可能不精确,故再次循环
                continue;
            }
            xnrm=snrm(n,x,itol);
            if (*err <= 0.5*xnrm) *err /= xnrm;
            else {
                *err=znrm/bnrm;              误差可能不精确,故再次循环
                continue;
            }
        }
        printf("iter=%4d err=%12.6f\n",*iter,*err);
    if (*err <= tol) break;
    }

    free_dvector(p,1,n);
    free_dvector(pp,1,n);
    free_dvector(r,1,n);
    free_dvector(rr,1,n);
    free_dvector(z,1,n);
    free_dvector(zz,1,n);
}
```

程序 **linbcg** 使用了下面这个小实用程序来计算向量的范数。

```
#include <math.h>

double snrm(unsigned long n, double sx[], int itol)      根据 itol 的值,计算向量 sx[1..n]的两个范数
{                                                         之一,被程序 linbcg 使用。
```

```
    unsigned long i,isamax;
    double ans;

    if (itol <= 3) {
        ans = 0.0;
        for (i=1;i<=n;i++) ans += sx[i]*sx[i];     向量大小的范数
        return sqrt(ans);
    } else {
        isamax=1;
        for (i=1;i<=n;i++) {                        最大元的范数
            if (fabs(sx[i]) > fabs(sx[isamax])) isamax=i;
        }
        return fabs(sx[isamax]);
    }
}
```

为使程序 **atimes** 和 **asolve** 的说明清楚些，我们在此列出简单的版本，它假定矩阵 **A** 以行索引稀疏格式存于某处。

```
extern unsigned long ija[];
extern double sa[];                                              矩阵存于某处

void atimes(unsigned long n, double x[], double r[], int itrnsp)
{
    void dsprsax(double sa[], unsigned long ija[], double x[], double b[],
        unsigned long n);                                        是 Sprsax 的双精度版本
    void dsprstx(double sa[], unsigned long ija[], double x[], double b[],
        unsigned long n);                                        是 Sprstx 的双精度版本
    if (itrnsp) dsprstx(sa,ija,x,r,n);
    else dsprsax(sa,ija,x,r,n);
}

extern unsigned long ija[];
extern double sa[];                                              矩阵存于某处

void asolve(unsinged long n, double b[], double x[], int itrnsp)
{
    unsigned long i;

    for(i=1;i<=n;i++) x[i]=(sa[i] != 0.0 ? b[i]/sa[i] : b[i]);   矩阵Ã是A的对角线部分，存于
                                                                  sa的前n个元素，因为该矩阵的
                                                                  转置对角元不变，故没有用标记
                                                                  itrnsp
}
```

**参考文献和进一步读物：**


Tewarson, R. P. 1973, *Sparse Matrices* (New York: Academic Press). [1]

Jacobs, D. A. H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), Chapter I. 3 (by J. K. Reid). [2]

George, A., and Liu, J. W. H. 1981, *Computer Solution of Large Sparse Positive Definite Systems* (Englewood Cliffs, NJ: Prentice-Hall). [3]

*NAG Fortran Library* (Numerical Algorithms Group, 256 Banbury Road, Oxford Ox27DE, U. K.). [4]

*IMSL Math/Library Users Manual* (IMSL Inc., 2500 CityWest Boulevard, Houston TX77042). [5]

Eisenstat, S. C., Gursky, M. C., Schultz, M. H., and Sherman, A. H. 1977 *Yale Sparse Matrix Package*, Technical Reports 112 and 114 (Yale University Department of Computer Science). [6]

Knuth, D. E. 1968, *Fundamental Algorithms*, vol. 1 of *The Art of Computer Programming* (Reading,

MA:Addison-Wesley), §2.2.6.[7]

Kincaid, D.R., Respess, J.R., Young, D.M., and Grimes, R.G. 1982, *ACM Transactions on Mathematical Software*, vol.8, pp.302~322. [8]

*PCGPAK User's Guide* (New Haven: Scientific Computing Associates, Inc.). [9]

Bentley, J. 1986, *Programming Pearls* (Reading, MA: Addison Wesley), §9. [10]

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), Chapters 4 and 10, particularly §§10.2~10.3. [11]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer Verlag), Chapter 8. [12]

Baker, L. 1991, *More C Tools for Scientists and Engineers* (New York: McGraw-Hill). [13]

Fletcher, R. 1976, in *Numerical Analysis Dundee* 1975, Lecture Notes in Mathematics, vol. 506, A. Dold and B Eckmann, eds. (Berlin: Springer-Verlag), pp. 73~89. [14]

Saad, Y., and Schulz, m. 1986, *SIAM Journal on Scientific and Statistical Computing*, vol. 7, pp. 856~869. [15]

## 2.8 范德蒙矩阵和托普雷兹矩阵

第2.4节中,三对角矩阵的情况是特殊处理的,因为这种特殊类型的线性系统可使求解运算仅为 $N$ 级,而不是一般线性问题所需的 $N^3$。如果存在这样的特殊类型,重要的是要了解它们。如果碰巧要求解含有特殊类型的问题,它所节省的计算量是很可观的。

这一节中,处理两种特殊形式的矩阵,它们可以在 $N^2$级运算内求解。虽然不象三对角形式那么好,但比一般情况要强多了(除了运算量外,这两种类型的矩阵没有别的共同之处)。第一种类型的矩阵称为**范德蒙矩阵**(Vandermonde Matrices),它经常出现在与多项式拟合、矩的分布重构以及其它一些内容有关的问题中。例如本书的在第3.5节中出现了范德蒙矩阵。第二种类型称为**托普雷兹矩阵**(Toeplitz Matrices),其很容易在解卷积和信号处理问题中遇到,本书在第13.7节中将遇到托普雷兹矩阵。

这并不是已知特殊形式矩阵的全部。**希尔伯特矩阵**(Hilbert Matrices)的元素形式为 $a_{ij}=1/(i+j-1)$,$i,j=1,\cdots,N$,可以用非常精确的整数算法来求逆。由于其数值情况很特殊,用其它别的方法来求逆是很困难的(祥情请参考[1])。已在第2.7节中讨论过的谢尔曼-莫里森(Sherman-Morrison)和伍德伯瑞(Wood bury)公式,有时可以用来把新的特殊形式转换成已有的形式。我们还没发现别的形式的矩阵象现在讨论的这两种那么常见。

### 2.8.1 范德蒙矩阵

$N\times N$ 的范德蒙矩阵完全由 $N$ 个数 $x_1,x_2,\cdots,x_N$ 所确定,这 $N^2$个元素按其整数次幂排列 $x_i^{j-1}$,$i,j=1,\cdots,N$。显然有两种可能的形式,它取决于是 $i$ 看作行、$j$ 看作列,还是反过来。对前者我们得到线性系统的方程如下:

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{N-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{N-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_N & x_N^2 & \cdots & x_N^{N-1} \end{bmatrix} \cdot \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_N \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \tag{2.8.1}$$

将矩阵乘法展开就会看到，该方程求解的未知系数 $c_i$，它们与 $N$ 对横坐标和纵坐标 $(x_j, y_j)$ 之间满足多项式关系。精确地求解该问题将在第3.5中讨论，那里给出的程序是用我们将要说明的方法来求解(2.8.1)的。

行列交换的方程组形式为：

$$\begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_N \\ x_1^2 & x_2^2 & \cdots & x_N^2 \\ & \cdots & & \\ x_1^{N-1} & x_2^{N-1} & \cdots & x_N^{N-1} \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \cdots \\ w_N \end{bmatrix} = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ \cdots \\ q_N \end{bmatrix} \tag{2.8.2}$$

将其展开会发现它与矩问题有关。给定 $N$ 个点 $x_i$ 的值，求未知的权 $w_i$，使其与前面的 $N$ 个矩的给定值 $q_i$ 相匹配。(想了解关于此问题更多的内容，请参考[3]。)这一节中给出的程序是用来求解 (2.8.2)的。

求解(2.8.1)和(2.8.2)的方法，都与拉格朗日的多项式插值公式密切相关，这将在后面的第3.1节中讨论。尽管如此，下面推导还是可以理解的：

设 $P_j(x)$ 为 $N-1$阶多项式，其定义为

$$P_j(x) = \prod_{\substack{n=1 \\ (n\neq j)}}^{N} \frac{x - x_n}{x_j - x_n} = \sum_{k=1}^{N} A_{jk} x^{k-1} \tag{2.8.3}$$

最后一个等式的意义为，将矩阵乘积展开成多项式形式并合并同类项，再把其系数定义为矩阵 $A_{ij}$的内容。

多项式 $P_j(x)$一般是 x 的函数。但会发现该函数的特殊之处，对所有 $x_i$，$i\neq j$ 时，其值为零；对 $x=x_j$，其值为1。换言之，

$$P_j(x_i) = \delta_{ij} = \sum_{k=1}^{N} A_{jk} x_i^{k-1} \tag{2.8.4}$$

式 (2.8.4)表明 $A_{jk}$是式(2.8.2)中元素 $x_i^{k-1}$ 所构成的矩阵的逆，这里 $x_i^{k-1}$ 的下标代表列索引。因此，式(2.8.2)的解就是该逆矩阵与右端项的乘积。

$$w_j = \sum_{k=1}^{N} A_{jk} q_k \tag{2.8.5}$$

至于该问题转置的形式(2.8.1)，根据转置矩阵的逆是逆矩阵的转置这一事实，所以

$$c_j = \sum_{k=1}^{N} A_{kj} y_k \tag{2.8.6}$$

在第3.5节中的程序来实现它。

留下的问题是，怎样找到一个好的方法，将式(2.8.3)中的各个单项乘出来，以便得到 $A_{jk}$的元素。这个问题很有必要写出来，这里想让读者自己读读程序，来看看它是如何解决的。先定义 $P(x)$：

$$P(x) \equiv \prod_{n=1}^{N} (x - x_n) \tag{2.8.7}$$

求出其系数，然后通过对一个另外的项做综合除法，得到 $P_j$ 的分子和分母项(关于综合除法详见第5.3节)。因为每一次这样的除法处理复杂度仅为 $N$ 阶，所以总的处理过程为 $N^2$阶。

必须说明，范德蒙问题由于其内在属性因是严重病态的。(在第5.8节中还要提到，其原因与要使切比雪夫拟合(Chebyshev)达到相当的精度一样：存在高阶多项式，其形式非常接近于零。因此，舍入误差会引入到这些多项式前几项的实际系数中。)因此始终采用双精度数来计算范德蒙问题是个好办法。

下面的求解方程组(2.8.2)的程序是由 G. B. Rybichi 提供的。

```
#include "nrutil.h"

void vander(double x[], double w[], double q[], int n)
```

求解范德蒙线性系统 $\sum_{i=1}^{N} x_i^{k-1} w_i = q_k$ $(k=1,\cdots,N)$。输入包括向量 x[1..n]和 q[1..n];输出为向量 w[1..n]

```
{
    int i,j,k,k1;
    double b,s,t,xx;
    double *c;

    c=dvector(1,n);
    if (n == 1) w[1]=q[1];
    else {
        for (i=1;i<=n;i++) c[i]=0.0;          初始化数组
        c[n] = -x[1];                          主多项式系数由递归得到
        for (i=2;i<=n;i++) {
            xx = -x[i];
            for (j=(n+1-i);j<=(n-1);j++) c[j] += xx*c[j+1];
            c[n] += xx;
        }
        for (i=1;i<=n;i++) {                   依次对每一个次因子
            xx=x[i];
            t=b=1.0;
            s=q[n];
            k=n;
            for (j=2;j<=n;j++) {               做综合除法
                k1=k-1;
                b=c[k]+xx*b;
                s += q[k1]*b;                  用右端项进行矩阵乘
                t=xx*t+b;
                k=k1;
            }
            w[i]=s/t;                          用分母除
        }
    }
    free_dvector(c,1,n);
}
```

## 2.8.2 托普雷兹矩阵

一个 $N\times N$ 的托普雷兹矩阵(Toeplitz Matrices)由给出的 $2N-1$ 个数 $R_k$ 组成,$k=-N+1,\cdots,-1,0,1,\cdots,N-1$。这些数作为矩阵的常量元素,按(左上至右下)对角线顺序排列:

$$\begin{bmatrix} R_0 & R_{-1} & R_{-2} & \cdots & R_{-(N-2)} & R_{-(N-1)} \\ R_1 & R_0 & R_{-1} & \cdots & R_{-(N-3)} & R_{-(N-2)} \\ R_2 & R_1 & R_0 & \cdots & R_{-(N-4)} & R_{-(N-3)} \\ \cdots & & & \cdots & & \\ R_{N-2} & R_{N-3} & R_{N-4} & \cdots & R_0 & R_{-1} \\ R_{N-1} & R_{N-2} & R_{N-3} & \cdots & R_1 & R_0 \end{bmatrix} \tag{2.8.8}$$

线性托普雷兹问题可写为

$$\sum_{j=1}^{N} R_{i-j} x_j = y_i \qquad (i=1,\cdots,N) \tag{2.8.9}$$

其中 $x_j$,$j=1,\cdots,N$ 为要求解的未知量。

如果对所有 $k$ 都有 $R_k=R_{-k}$,则该托普雷兹阵是对称的。莱文逊(Levinson)设计了一个求对称的托普雷兹问题的快速算法,**边界法**。它是一个递归过程,用来求解 $M$ 维的托普雷兹问题

$$\sum_{j=1}^{M} R_{i-j} x_j^{(M)} = y_i \qquad (i=1,\cdots,M) \tag{2.8.10}$$

依次取 $M=1,2,\cdots$ 直到 $M=N$,便可求得最后要求的结果。向量 $x_j^{(M)}$ 是第 $M$ 次得到的结果,只有当 $M$ 达到 $N$ 时,才是要求的解。

莱文逊的方法在标准的教科书中(例如[5])都有详细的介绍;但这种方法推广到**非对称阵**的情况,却鲜为人知。这里给出一个由 Rybicki 提供的推导。

在下面由第 M 步到第 $M+1$ 步的推导中,会发现递推解 $x^{(M)}$ 的变化如下:

$$\sum_{j=1}^{M} R_{i-j} x_j^{(M)} = y_i \qquad (i=1,\cdots,M) \tag{2.8.11}$$

变成

$$\sum_{j=1}^{M} R_{i-j} x_j^{(M+1)} + R_{i-(M+1)} x_{M+1}^{(M+1)} = y_i \qquad i=1,\cdots,M+1 \tag{2.8.12}$$

消去 $y_i$,得:

$$\sum_{j=1}^{M} R_{i-j}\left(\frac{x_j^{(M)}-x_j^{(M+1)}}{x_{M+1}^{(M+1)}}\right) = R_{i-(M+1)} \qquad i=1,\cdots,M \tag{2.8.13}$$

或者将 $i\to M+1-i$,并将 $j\to M+1-j$

$$\sum_{j=1}^{M} R_{j-i} G_j^{(M)} = R_{-i} \tag{2.8.14}$$

其中

$$G_j^{(M)} \equiv \frac{x_{M+1-j}^{(M)} - x_{M+1-j}^{(M+1)}}{x_{M+1}^{(M+1)}} \tag{2.8.15}$$

将其改写为:

$$x_{M+1-j}^{(M+1)} = x_{M+1-j}^{(M)} - x_{M+1}^{(M+1)} G_j^{(M)} \qquad j=1,\cdots,M \tag{2.8.16}$$

因此,如果由此递推式求得 $M$ 阶的值 $x^{(M)}$ 和 $G^{(M)}$,以及一个 $M+1$阶的值 $x_{M+1}^{(M+1)}$,则其余所有的 $x_j^{(M+1)}$ 便可求得。幸运的是,值 $x_{M+1}^{(M+1)}$ 可以如下求得:在方程(2.8.12)中,令 $i=M+1$,

$$\sum_{j=1}^{M} R_{M+1-j} x_j^{(M+1)} + R_0 x_{M+1}^{(M+1)} = y_{M+1} \tag{2.8.17}$$

对未知的 $M+1$阶量 $x_j^{(M+1)}$,可以用 $G$ 中的低阶量来替换。由

$$G_{M+1-j}^{(M)} = \frac{x_j^{(M)} - x_j^{(M+1)}}{x_{M+1}^{(M+1)}} \tag{2.8.18}$$

运算的结果为

$$x_{M+1}^{(M+1)} = \frac{\sum_{j=1}^{M} R_{M+1-j} x_j^{(M)} - y_{M+1}}{\sum_{j=1}^{M} R_{M+1-j} G_{M+1-j}^{(M)} - R_0} \tag{2.8.19}$$

剩下的唯一问题是求 $G$ 的递推关系。然而在构造递推关系之前,我们必须指出,对原来的非对称矩阵的线性问题,有两个完全不同的解集,即右端项解(我们已讨论的)和左端项解 $z_i$。左端项解的形式差别仅在于要处理的方程为

$$\sum_{j=1}^{M} R_{j-i} z_j^{(M)} = y_i \qquad i=1,\cdots,M \tag{2.8.20}$$

然后,对该集合做类似的操作,得

$$\sum_{j=1}^{M} R_{i-j} H_j^{(M)} = R_i \tag{2.8.21}$$

其中

$$H_j^{(M)} = \frac{z^{(M+1)}_{M+1-j} - z^{(M)}_{M+1-j}}{z^{(M+1)}_{M+1}} \tag{2.8.22}$$

(与式2.8.14～2.8.15比较)。提及左端项解的原因在于，在方程(2.8.21)中，要是用 $y_i \to R_i$ 替换右端项，则 $H_j$ 与 $x_i$ 满足相同的方程，因此，由方程(2.8.19)我们可以很快推出

$$H^{(M+1)}_{M+1} = \frac{\sum_{j=1}^{M} R_{M+1-j} H_j^{(M)} - R_{M+1}}{\sum_{j=1}^{M} R_{M+1-j} G^{(M)}_{M+1-j} - R_0} \tag{2.8.23}$$

如果使用替换 $y_i \to R_{-i}$，并用相同的符号表示，则 $G$ 满足 $Z$ 所表示的相同的方程，即

$$G^{(M+1)}_{M+1} = \frac{\sum_{j=1}^{M} R_{j-M-1} G_j^{(M)} - R_{-M-1}}{\sum_{j=1}^{M} R_{j-M-1} H^{(M)}_{M+1-j} - R_0} \tag{2.8.24}$$

同理，变换方程(2.8.16)以及 Z 所表示方程的对应量，得最后的方程为

$$\begin{aligned} G_j^{(M+1)} &= G_j^{(M)} - G^{(M+1)}_{M+1} H^{(M)}_{M+1-j} \\ H_j^{(M+1)} &= H_j^{(M)} - H^{(M+1)}_{M+1} G^{(M)}_{M+1-j} \end{aligned} \tag{2.8.25}$$

现在，来看看方程的初值

$$x_1^{(1)} = y_1/R_0 \qquad G_1^{(1)} = R_{-1}/R_0 \qquad H_1^{(1)} = R_1/R_0 \tag{2.8.26}$$

我们可以递推下去，在第 $M$ 步求解过程中，由方程(2.8.23)和(2.8.24)求得 $H^{(M+1)}_{M+1}$、$G^{(M+1)}_{M+1}$，然后由方程(2.8.25)求得另外的 $H^{(M+1)}$ 和 $G^{(M+1)}$ 的内容，由此，向量 $\mathbf{x}^{(M+1)}$ 和/或 $\mathbf{z}^{(M+1)}$ 便很容易求出了。

下面的程序完成此功能，它将(2.8.25)的第二个方程形式变为

$$H^{(M+1)}_{M+1-j} = H^{(M)}_{M+1-j} - H^{(M+1)}_{M+1} G_j^{(M)} \tag{2.8.27}$$

这样计算可以"同址"进行。

注意如果 $R_0=0$，则上述算法失败。事实上，由于边界法没有采用主元素法，如果将原来的托普雷兹矩阵中主对角线上的小元素化成零，算法也会失败(跟第2.4节中讨论的三对角阵算法对比一下)。如果算法失败，矩阵并不一定是奇异的——可能必须用一个慢一点但更为通用的算法，例如，使用主元素法的 LU 分解来求解该问题。

该程序实现方程(2.8.23)～(2.8.27)，也是由 Rybicki 提供的。注意，程序中的 r[n+j]与上面的 $R_j$ 相当，因此，数组 r 的下标变化范围为 $1\sim 2N-1$。

```
#include "nrutil.h"
#define FREERETURN {free_vector(h,1,n);free_vector(g,1,n);return;}

void toeplz(float r[], float x[], float y[], int n)
```

求解托普雷兹问题 $\sum_{j=1}^{N} R_{(N+i-j)} x_j = y_i \ (i=1,\ldots,N)$。托普雷兹矩阵不必是对称的。y[1..n]和 r[1..2*n-1]是各自长度为n和2*n-1的输入矩阵，x[1..n]是输出矩阵。

```
{
    int j,k,m,m1,m2;
    float pp,pt1,pt2,qq,qt1,qt2,sd,sgd,sgn,shn,sxn;
    float *g,*h;

    if (r[n] == 0.0) nrerror("toeplz-1 singular principal minor");
    g=vector(1,n);
    h=vector(1,n);
    x[1]=y[1]/r[n];                 为递推初始化
    if (n == 1) FREERETURN
```

```
    g[1]=r[n-1]/r[n];
    h[1]=r[n+1]/r[n];
    for (m=1;m<=n;m++) {            递推的主循环
        m1=m+1;
        sxn = -y[m1];               计算x的分子和分母项
        sd = -r[n];
        for (j=1;j<=m;j++) {
            sxn += r[n+m1-j]*x[j];
            sd += r[n+m1-j]*g[m-j+1];
        }
        if (sd == 0.0) nrerror("toeplz-2 singular principal minor");
        x[m1]=sxn/sd;                                        由此得到x
        for (j=1;j<=m;j++) x[j] -= x[m1]*g[m-j+1];
        if (m1 == n) FREERETURN
        sgn = -r[n-m1];                         计算G和H的分子和分母项
        shn = -r[n+m1];
        sgd = -r[n];
        for (j=1;j<=m;j++) {
            sgn += r[n+j-m1]*g[j];
            shn += r[n+m1-j]*h[j];
            sgd += r[n+j-m1]*h[m-j+1];
        }
        if (sd == 0.0 || sgd == 0.0) nrerror("toeplz-3 singular principal minor");
        g[m1]=sgn/sgd;              由此得到G和H
        h[m1]=shn/sd;
        k=m;
        m2=(m+1) >> 1;
        pp=g[m1];
        qq=h[m1];
        for (j=1;j<=m2;j++) {
            pt1=g[j];
            pt2=g[k];
            qt1=h[j];
            qt2=h[k];
            g[j]=pt1-pp*qt2;
            g[k]=pt2-pp*qt1;
            h[j]=qt1-qq*pt2;
            h[k--]=qt2-qq*pt1;
        }
    }                               返回进行下一次递推
    nrerror("toeplz - should not arrive here!");
}
```

如果在读者工作中,遇到求解巨大的托普雷兹系统,你将会发现有一些声称为"新的、快速"的算法,它只要求 $N(\log N)^2$ 阶操作,而莱文孙算法需 $N^2$ 阶操作。但这些算法非常复杂,可参考本奇(Bunch)[6]和德·胡格(de Hoog)[7]的文献,它们给出这些算法的有关内容。

## 参考文献和进一步读物:


Forsythe, G. E., and Moler, C. B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, NJ: Prentice Hall), §19. [1]

Westlake, J. R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley). [2]

von Mises, R. 1964, *Mathematical Theory of Probability and Statistics* (New York: Academic Press), pp. 394ff. [3]

Levinson, N., Appendix B of N. Wiener, 1949, *Extrapolation, Interpolation and Smoothing of Stationary Time Series* (New York: Wiley). [4]

Robinson, E. A., and Treitel, S. 1980, *Geophysical Signal Analysis* (Englewood Cliffs, NJ: Prentice-Hall), pp. 163ff. [5]

Bunch, J. R. 1985, *SIAM Journal on Scientific and Statistical Computing*, vol. 6, pp. 349-364. [6]
de Hoog, F. 1987, *Linear Algebra and its Applications*, vol. 88/89, pp. 123-138. [7]

## 2.9 乔莱斯基分解

如果一个方阵 **A** 恰好是对称的和正定的，则它有一特殊的更有效的三角分解。**对称的**意味着 $a_{ij}=a_{ji}$，对 $i,j=1,\cdots,N$；**正定的**意味着

$$\mathbf{v}\cdot\mathbf{A}\cdot\mathbf{v}>0 \quad \text{对所有向量 } \mathbf{v} \tag{2.9.1}$$

（在第11章中，我们将会看到正定的等价解释为，**A** 的所有特征值均为正数。）对称正定的矩阵相当特殊，它们在某些应用中会经常出现，故最好知道其特殊的分解，称作**乔莱斯基**（cholesky）**分解**。使用这种方法求解线性方程组时，差不多要比别的方法快两倍。

乔莱斯基分解并不是寻找任意的下三角和上三角因子 **L** 和 **U**，而是构造一个下三角矩阵 **L**，其转置本身作为上三角部分。换言之，我们可以把等式(2.3.1)换成

$$\mathbf{L}\cdot\mathbf{L}^T=\mathbf{A} \tag{2.9.2}$$

该分解有时被引用为，对矩阵 **A**"取平方根"。$\mathbf{L}^T$ 的元素与 **L** 的元素的关系自然是

$$L_{ij}^T=L_{ji} \tag{2.9.3}$$

写出方程(2.9.2)的元素，很容易得到与方程(2.3.12)－(2.3.13)类似的结果。

$$L_{ii}=\left(a_{ii}-\sum_{k=1}^{i-1}L_{ik}^2\right)^{1/2} \tag{2.9.4}$$

及

$$L_{ji}=\frac{1}{L_{ii}}\left(a_{ij}-\sum_{k=1}^{i-1}L_{ik}L_{jk}\right) \qquad j=i+1,i+2,\cdots,N \tag{2.9.5}$$

如果按顺序 $i=1,2,\cdots,N$ 使用方程(2.9.4)和(2.9.5)，就会发现右端出现的 $L$ 在需要时已经确定了。还有，仅有 $j\geq i$ 的元素 $a_{ij}$ 被引用(因为 **A** 是对称的，这些包含了全部的信息)。然后，很方便地就可把因子 **L** 覆盖写入 **A** 的子对角线部分(下三角，但不包括对角线)，而保留输入的 **A** 的上三角的值不变。因而只需一个长度为 $N$ 的附加向量存储 **L** 的对角线部分。其运算次数为 $N^3/6$ 次内层循环(包括一次乘，一次减)及 $N$ 次平方根运算。前面已提到过，这差不多要比 **A** 的 $LU$ 分解快两倍(因它没有考虑矩阵的对称性)。

一个直接了当的实现为

```
#include <math.h>

void choldc (float **a, int n, float p[])
    对给出的正定对称矩阵a[1..n][1..n]，本程序构造其乔莱斯基分解，A=L·L^T。输入量仅需给出a的上三角部分，
    而且它不被程序修改。乔莱斯基因子L返回于a的下三角部分，但其对角线元素返回到p[1..n]。
{
    void nrerror(char error_text[]);
    int i,j,k;
    float sum;

    for (i=1;i<=n;i++) {
        for (j=i;j<=n;j++) {
            for (sum=a[i][j],k=i-1;k>=1;k--) sum -= a[i][k]*a[j][k];
            if (i == j) {
                if (sum <= 0.0)                    由于舍入误差的缘故，a不是正定的
                    nrerror("choldc failed");
                p[i]=sqrt(sum);
```

```
            } else a[j][i]=sum/p[i];
        }
    }
}
```

此时,读者可能会想是否需要找主元。令人高兴的是,乔莱斯基分解在数值上极为稳定。而程序 **choldc** 的失败,只能表明矩阵 **A**(或是由于舍入误差的缘故,有另一个与其非常接近的矩阵)不是正定的。事实上,程序 **choldc** 是一种检测一个对称矩阵是否是正定的很有效的方法(对于这种应用,可以把对 nrerror 的调用换成某种信号缓和些的方法)。

一旦矩阵被分解了,可通过回代过程用三角因子来求解线性方程。这种直接的应用为

```
void cholsl (float **a, int n, float p[], float b[], float x[])
```

求解 n 维线性方程组 **A**·**x**=**b**,其中 a 为正定对称矩阵。输入量 a[1..n][1..n]和 p[1..n]是程序 **choldc** 输出的结果。只有 a 的下三角部分被访问。b[1..n]输入量是右端项向量。解向量返回至 x[1..n]中。a,n 和 p 没有被修改,可留作不同右端项 b 的连续调用。b 也不被修改,除非调用时将 b 和 x 设成一个,这种情况是允许的。

```
{
    int i,k;
    float sum;

    for (i=1;i<=n;i++) {                                        求解 L·y=b,将 y 存于 x
        for (sum=b[i],k=i-1;k>=1;k--) sum -= a[i][k]*x[k];
        x[i]=sum/p[i];
    }
    for (i=n;i>=1;i--) {                                        求解 L^T·x=y
        for (sum=x[i],k=i+1;k<=n;k++) sum -= a[k][i]*x[k];
        x[i]=sum/p[i];
    }
}
```

**choldc** 和 **cholsl** 的一个典型应用是,求描述数据适合某种模型的协方差矩阵的逆,参见如第15.6节。在这里及其它许多应用中,经常需要 $\mathbf{L}^{-1}$。该矩阵的下三角部分可以高效地从 **choldc** 的输出中求得:

```
    for (i=1; i<=n; i++) {
        a[i][i]=1.0/p[i];
        for (j=i+1; j<=n; j++) {
                  sun=0.0;
                  for (k=i; k<j; k++) sum -= a[j][k]*a[k][i];
                  a[j][i]=sum/p[j];
        }
    }
```

## 2.10 *QR* 分解

还有另一种矩阵的因式分解,有时非常有用,即所谓的 *QR* 分解:

$$\mathbf{A}=\mathbf{Q}\cdot\mathbf{R} \tag{2.10.1}$$

此处 **R** 是上三角的,而 **Q** 是正交的,即

$$\mathbf{Q}^T\cdot\mathbf{Q}=\mathbf{1} \tag{2.10.2}$$

其中 $\mathbf{Q}^T$ 为 **Q** 的转置。尽管对一般的长方形矩阵,这种分解都存在,但我们还是限定为维数为 $N\times N$ 的方阵的情况。

跟我们已遇到过的其它矩阵分解(*LU*,SVD,Cholesky)一样,*QR* 分解也可用于求解线性方程系统。欲求解

$$\mathbf{A}\cdot\mathbf{x}=\mathbf{b} \tag{2.10.3}$$

首先构造 $Q^T \cdot b$,然后通过回代过程求解

$$R \cdot x = Q^T \cdot b \tag{2.10.4}$$

因为 $QR$ 分解的运算操作差不多是 $LU$ 分解的两倍,所以它不用于典型的线性方程系统。然而,我们会遇到一些特殊情况,$QR$ 分解是可选的方法。

标准的 $QR$ 分解算法包括连续的豪斯贺德变换(在后面第11.2节讨论)。我们写一个豪斯贺德矩阵,格式为 $1 - u \otimes u/c$,其中 $c = \frac{1}{2} u \cdot u$。合适的豪斯贺德矩阵用在给定的矩阵上,可以使位于某选定元素之下的矩阵列元素全部零化。因此,我们设第一个豪斯贺德矩阵 $Q_1$,使 A 的第一列第一个元素之下的元素全化为零。同样,$Q_2$ 使第二列第二个元素之下的元素全化为零,如此直到 $Q_{n-1}$。所以

$$R = Q_{n-1} \cdots Q_1 \cdot A \tag{2.10.5}$$

因为豪斯贺德矩阵是正交的,

$$Q = (Q_{n-1} \cdots Q_1)^{-1} = Q_1 \cdots Q_{n-1} \tag{2.10.6}$$

在大部分应用中,我们不需要显式地形成 Q,只需将其存成(2.10.6)的因子的形式,除非矩阵 A 非常接近奇异,否则不必求主元。[1]中给出了一种通用的,对长方形矩阵的 $QR$ 算法,它包括了求主元、对方阵。一种实现程序如下:

```
#include  <math.h>
#include "nrutil.h"

void qrdcmp (float **a, int n, float *c, float *d, int *sing)
```

构造 a[1..n][1..n]的 $QR$ 分解。上三角矩阵 R 返回到 a 的上三角部分,R 的对角元返回到 d[1..n]。正交矩阵 Q 被表示成 $n-1$ 个豪斯贺德矩阵 $Q_1 \cdots Q_{n-1}$ 的积的形式,其中 $Q_j = 1 - u_j \otimes u_j / c_j$。对 $i = 1, \cdots, j-1$,$u_j$ 的第 i 个元素为零,非零元返回在 a[i][j]中,$i = j, \cdots, n$。如果分解中出现奇异的情况,sing 值返回为真(1),但本例中,应完成分解;否则返回为假(0)。

```
{
    int i,j,k;
    float scale=0.0,sigma,sum,tau;

    *sing=0;
    for (k=1;k<n;k++) {
        for (i=k;i<=n;i++) scale=FMAX(scale,fabs(a[i][k]));
        if (scale == 0.0) {                         奇异情况
            *sing=1;
            c[k]=d[k]=0.0;
        } else {                                    形成 Qk 及 Qk · A.
            for (i=k;i<=n;i++) a[i][k] /= scale;
            for (sum=0.0,i=k;i<=n;i++) sum += SQR(a[i][k]);
            sigma=SIGN(sqrt(sum),a[k][k]);
            a[k][k] += sigma;
            c[k]=sigma*a[k][k];
            d[k] = -scale*sigma;
            for (j=k+1;j<=n;j++) {
                for (sum=0.0,i=k;i<=n;i++) sum += a[i][k]*a[i][j];
                tau=sum/c[k];
                for (i=k;i<=n;i++) a[i][j] -= tau*a[i][k];
            }
        }
    }
    d[n]=a[n][n];
    if (d[n] == 0.0) *sing=1;
}
```

下面的程序 **qrsolv** 用于求解线性系统。在许多应用中,仅需要算法的(2.10.4)部分,因此,我们将其分离成单独的程序 *rsolv*。

```
void qrsolv(float **a, int n, float c[], float d[], float b[])
```

求解 n 维线性方程组 $\mathbf{A}\cdot\mathbf{x}=\mathbf{b}$。输入量 a[1..n][1..n],c[1..n]和 d[1..n]是程序 **qrdcmp** 的输出结果,而且不被修改。输入量 b[1..n]为右端向量,并被输出的解向量所覆盖写入。

```
{
    void rsolv(float **a, int n, float d[], float b[]);
    int i,j;
    float sum,tau;

    for (j=1;j<n;j++) {                                              形式 Q^T·b
        for (sum=0.0,i=j;i<=n;i++) sum += a[i][j]*b[i];
        tau=sum/c[j];
        for (i=j;i<=n;i++) b[i] -= tau*a[i][j];
    }
    rsolv(a,n,d,b);                                                  求解 R·x=Q^T·b
}
```

```
void rsolv(float **a, int n, float d[], float b[])
```

求解 n 维线性方程组 $\mathbf{R}\cdot\mathbf{x}=\mathbf{b}$,其中 $\mathbf{R}$ 为存于 a 和 d 的上三角矩阵,输入量 a[1..n]和 d[1..n]是程序 **qrdcmp** 的输出结果,而且不被修改。输入量 b[1..n]为右端向量,并被输出的解向量所覆盖写入。

```
{
    int i,j;
    float sum;

    b[n] /= d[n];
    for (i=n-1;i>=1;i--) {
        for (sum=0.0,j=i+1;j<=n;j++) sum += a[i][j]*b[j];
        b[i]=(b[i]-sum)/d[i];
    }
}
```

关于如何使用 *QR* 分解构造正交基并求解最小平方的问题,可详细参考[2]。(对这种用途,我们喜欢用第2.6节的 SVD 方法,因为它对病态情况具有较强的诊断能力。)

## 2.10.1 更新 QR 分解

有些数值算法包含求解一连串的线性系统,而每个系统与前一个仅有很小的差别。不必每次使用 $O(N^3)$次运算从零开始求解,我们经常可以在 $O(N^2)$次运算内更新矩阵因子,再用这个新的因子来求解下个线性方程组。由于找主元的原因,*LU* 分解更新很复杂。然而 QR 分解对一种很常见的更新却是非常容易的,

$$\mathbf{A}\rightarrow\mathbf{A}+\mathbf{s}\otimes\mathbf{t} \tag{2.10.7}$$

(与方程2.7.1比较)。实际上,用其等价的形式来计算更方便

$$\mathbf{A}=\mathbf{Q}\cdot\mathbf{R}\quad\rightarrow\quad\mathbf{A}'=\mathbf{Q}'\cdot\mathbf{R}'=\mathbf{Q}\cdot(\mathbf{R}+\mathbf{u}\otimes\mathbf{v}) \tag{2.10.8}$$

可以在等式(2.10.7)和(2.10.8)之间,反复使用 $\mathbf{Q}$ 是正交的这一事实,得到

$$\mathbf{t}=\mathbf{v}\quad\text{及}\quad\mathbf{s}=\mathbf{Q}\cdot\mathbf{u}\quad\text{或}\quad\mathbf{u}=\mathbf{Q}^T\cdot\mathbf{s} \tag{2.10.9}$$

算法[2]有两个阶段。第一阶段,我们应用 $N-1$次雅可比旋转(第11.1节),将 $\mathbf{R}+\mathbf{u}\otimes\mathbf{v}$ 归约成上海森伯格(Hessenberg)形式。另 $N-1$次雅可比旋转将该上海森伯格矩阵转换成新的上三角矩阵 $\mathbf{R}'$。矩阵 $\mathbf{Q}'$ 为 $\mathbf{Q}$ 与$2(N-1)$次雅可比旋转的乘积形式。在应用中通常需要 $\mathbf{Q}^T$,该算法很容易将 $\mathbf{Q}$ 换成 $\mathbf{Q}^T$ 的形式计算。

```
#include <math.h>
#include "nrutil.h"
void qrupdt(float **r, float **qt, int n, float u[], float v[])
```

对某个给出的 n×n 矩阵的 *QR* 分解,计算矩阵 $\mathbf{Q}\cdot(\mathbf{R}+\mathbf{u}\otimes\mathbf{v})$的 *QR* 分解。这些量的维数为 r[1..n][1..n]

[1..n][1..n]、u[1..n]及v[1..n]，注意$Q^T$为输入量，并返回在qt中.

```
{
    void rotate(float **r, float **qt, int n, int i, float a, float b);
    int i,j,k;

    for (k=n;k>=1;k--) {                    寻找最大的 k 使 u[k] ≠ 0.
        if (u[k]) break;
    }
    if (k < 1) k=1;
    for (i=k-1;i>=1;i--) {                   将 R + u ⊗ v 转换成上海森伯格形式
        rotate(r,qt,n,i,u[i],-u[i+1]);
        if (u[i] == 0.0) u[i]=fabs(u[i+1]);
        else if (fabs(u[i]) > fabs(u[i+1]))
            u[i]=fabs(u[i])*sqrt(1.0+SQR(u[i+1]/u[i]));
        else u[i]=fabs(u[i+1])*sqrt(1.0+SQR(u[i]/u[i+1]));
    }
    for (j=1;j<=n;j++) r[1][j] += u[1]*v[j];
    for (i=1;i<k;i++)                        将上海森伯格矩阵转换成上三角形式
        rotate(r,qt,n,i,r[i][i],-r[i+1][i]);
}
```

```
#include <math.h>
#include "nrutil.h"

void rotate(float **r, float **qt, int n, int i, float a, float b)
```

对给定的矩阵r[1..n][1..n]和qt[1..n][1..n]按每个矩阵的i行和i+1行进行雅可比旋转，a和b为旋转参数：

$$\cos\theta=\frac{a}{\sqrt{a^2+b^2}},\quad \sin\theta=\frac{b}{\sqrt{a^2+b^2}}。$$

```
{
    int j;
    float c,fact,s,w,y;

    if (a == 0.0) {                 避免不必要的上溢或下溢
        c=0.0;
        s=(b >= 0.0 ? 1.0 : -1.0);
    } else if (fabs(a) > fabs(b)) {
        fact=b/a;
        c=SIGN(1.0/sqrt(1.0+(fact*fact)),a);
        s=fact*c;
    } else {
        fact=a/b;
        s=SIGN(1.0/sqrt(1.0+(fact*fact)),b);
        c=fact*s;
    }
    for (j=i;j<=n;j++) {            r 预乘以雅可比旋转
        y=r[i][j];
        w=r[i+1][j];
        r[i][j]=c*y-s*w;
        r[i+1][j]=s*y+c*w;
    }
    for (j=1;j<=n;j++) {            qt 预乘以雅可比旋转
        y=qt[i][j];
        w=qt[i+1][j];
        qt[i][j]=c*y-s*w;
        qt[i+1][j]=s*y+c*w;
    }
}
```

我们将在第9.7节中，使用QR分解及其更新部分，

参考文献和进一步读物：

Wilkinson, J. H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag), Chapter I/8. [1]

Golub, G. H., and Van Loan, C. F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §§5.2, 5.3, 12.6. [2]

## 2.11 矩阵求逆是 $N^3$ 阶运算吗?

在结束本章之前,我们来做个算术游戏,进一步探讨矩阵求逆的问题。从一个看似简单的问题开始:

对2×2的矩阵进行矩阵乘,要做多少次乘法运算.

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \tag{2.11.1}$$

八次,对吗?明确地写下来即:

$$\begin{aligned} c_{11} &= a_{11} \times b_{11} + a_{12} \times b_{21} \\ c_{12} &= a_{11} \times b_{12} + a_{12} \times b_{22} \\ c_{21} &= a_{21} \times b_{11} + a_{22} \times b_{21} \\ c_{22} &= a_{21} \times b_{12} + a_{22} \times b_{22} \end{aligned} \tag{2.11.2}$$

你相信有人能仅用七次乘法,就写出求 $C$ 的公式吗?

事实上,这套公式是由斯特拉森(Strassen)发现的,这些公式为:

$$\begin{aligned} Q_1 &\equiv (a_{11} + a_{22}) \times (b_{11} + b_{22}) \\ Q_2 &\equiv (a_{21} + a_{22}) \times b_{11} \\ Q_3 &\equiv a_{11} \times (b_{12} - b_{22}) \\ Q_4 &\equiv a_{22} \times (-b_{11} + b_{21}) \\ Q_5 &\equiv (a_{11} + a_{12}) \times b_{22} \\ Q_6 &\equiv (-a_{11} + a_{21}) \times (b_{11} + b_{12}) \\ Q_7 &\equiv (a_{12} - a_{22}) \times (b_{21} + b_{22}) \end{aligned} \tag{2.11.3}$$

由此得:

$$\begin{aligned} c_{11} &= Q_1 + Q_4 - Q_5 + Q_7 \\ c_{21} &= Q_2 + Q_4 \\ c_{12} &= Q_3 + Q_5 \\ c_{22} &= Q_1 + Q_3 - Q_2 + Q_6 \end{aligned} \tag{2.11.4}$$

这么做有什么用呢?虽然比方程(2.11.2)中少了一次乘法,但却多了许多加法和减法,还看不出来得到了什么好处。但注意到式(2.11.3)中,$a$ 和 $b$ 并不互相引用,因此当 $a$ 和 $b$ 本身又是矩阵时,方程组(2.11.3)和(2.11.4)就有效。对非常大的两个矩阵相乘问题(对某个整数 $m$,维数 $N=2^m$),可将其递归地分解为四分之一,十六分之一等等。关键是要注意到:节省后的运算量不仅是一个"7/8"的因子;而递归的每一级都是这个因子,它总共将矩阵乘的运算量由 $N^3$ 级减少到 $N^{\log_2 7}$。

那么式(2.11.3)～(2.11.4)中额外的加法怎么算呢?它会不会超过了由于少算乘法带来的好处呢?对大的 $N$ 值,可证明式(2.11.3)～(2.11.4)中蕴含的加法次数是乘法次数的

六倍。但是，如果 $N$ 非常大，该常数因子就不能与从 $N^3$ 降到 $N^{\log_2 7}$ 的指数变化相提并论了。

在进行“快速”矩阵乘法的研究中，斯特拉森还得到了关于矩阵求逆的令人吃惊的结果。假定矩阵

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad 和 \quad \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \tag{2.11.5}$$

互为逆矩阵。则 $c$ 可由 $a$ 按下列操作得到：

$$\begin{aligned}
R_1 &= \text{Inverse}\ (a_{11}) \\
R_2 &= a_{21} \times R_1 \\
R_3 &= R_1 \times a_{12} \\
R_4 &= a_{21} \times R_3 \\
R_5 &= R_4 - a_{22} \\
R_6 &= \text{Inverse}\ (R_5) \\
c_{12} &= R_3 \times R_6 \\
c_{21} &= R_6 \times R_2 \\
R_7 &= R_3 \times c_{21} \\
c_{11} &= R_1 - R_7 \\
c_{22} &= -R_6
\end{aligned} \tag{2.11.6}$$

在式(2.11.6)中，“逆(Inverse)”运算有两次。该运算被解释为，若 $a$ 和 $c$ 为数值，则求其倒数；若 $a$ 和 $c$ 本身为子矩阵，则求其逆矩阵。想象一下，对一个维数 $N=2^m$ 的非常大的矩阵求逆时，用二分法递归求解的情况。在每一步递归过程中，都要使求逆运算的次数加倍，但这仅意味着总共做 $N$ 次除法！所以在递归使用式(2.11.6)时，除法并不是主要运算。事实上，等式(2.11.6)主要的运算是它的六次乘法。因为这些乘法可用复杂度为 $N^{\log_2 7}$ 的算法实现，故矩阵求逆也能如此！

这很有趣，但我们来看看其实用性：如果估算一下，要多大的 N 才能体现出指数 3 和指数 $\log_2 7=2.807$ 的差别来，实际上这就是浪费笔墨了。由于递归斯特拉森算法的内在复杂性，就会发现 $LU$ 分解法并不面临过时的危险。

另一方面，如果读者对这种游戏感兴趣，可以再试试这些：(1)能仅用三次实数乘法，就求出两个复数$(a+ib)$和$(c+id)$的乘积吗？[答案见第5.4节]。(2)能在计算一般 $x$ 的四次多项式时，对许多不同的 $x$ 值，每次仅用三次乘法就能求出其值吗？[答案见第5.3节]

参考文献和进一步读物：


Strassen，V.1969，*Numerische Mathematik*，vol. 13，pp. 354～356.[1]

# 第三章　内插法和外推法

## 3.0　引言

有时，我们仅知道函数 $f(x)$ 在一系列点 $x_1, x_2, \cdots, x_N$ 上的数值(假如 $x_1 < \cdots < x_N$)，但并没有 $f(x)$ 的解析式来求得任意一点的数值。例如，$f(x_i)$ 可能来源于一些物理测量值，或来源于不能用简单函数形式表示的冗长的数值运算。通常 $x_i$ 是等间距的，但不是必须如此。

现在的任务是，画一条通过(可能越过)$x_i$ 的平滑曲线，来对任意 $x$ 估算 $f(x)$ 的值。如果求的 $x$ 处于 $x_i$ 的最大值与最小值之间，则称之为**插值法**(内插法)；如果 $x$ 超出此范围则称为**外推法**，后者冒险性更大一些。(许多股票市场分析家可以证明)。

内插法和外推法是在已知点之间或越过已知点，用某些可能的函数形式来模拟此函数。函数的形式应该是很常见的，而且能够近似于应用中可能遇到的几大类函数。函数中最常用的形式为多项式(第3.1节)。有理函数(多项式的商)也被证明是非常有用的(第3.2节)。正弦和余弦等三角函数应用于**三角插值法**及相关的傅里叶方法中，我们将它推迟到第十二章和第十三章中讨论。

在内容全面的数学文献中，都有关于何类函数用作插值函数能很好地进行估算的定理。遗憾的是，这些定理通常几乎完全没什么用；如果我们对函数很熟悉，能够自如地应用有关的定理，也就不会处于必须根据数值表来内插函数的可怜处境了。

内插法与**函数逼近**有关，但也有不同之处。函数逼近的任务是，找一个近似的(但容易计算的)函数用以取代原来较复杂的函数。在插值法情况中，是要在不由自己选择的点上给出函数 $f$ 的值。而对于函数逼近，是为了得到近似函数，并允许在任意设定点计算出函数 $f$ 的值。我们将在第五章中讨论函数逼近问题。

可以很容易地构造一些病态函数使内插法失败。例如，考虑函数

$$f(x) = 3x^2 + \frac{1}{\pi^4}\ln[(\pi - x)^2] + 1 \tag{3.0.1}$$

它除了 $x=\pi$ 之外都有定义，而 $x=\pi$ 时无定义，其它情况，值有正有负。而这函数在任何基于数值 $x=3.13, 3.14, 3.15, 3.16$ 的插值法，都肯定在 $x=3.1416$ 处得到一个错误的解，尽管通过这五个点所画的曲线确实相当平滑！(用计算器试试看。)

因为这种病态函数可能隐藏在任何地方，所以要求内插法和外推法的程序，能提供自身的错误评估。这样的错误评估自然不会是简单透顶的。我们可能有个函数，它丢弃掉在已知两点之间那些相差太远或是不符合要求的点，其理由只有函数构造者才清楚。内插法总是假定进行内插的函数有一定程度的平滑性，在此假定下，偏离平滑也应能被检测出来。

概念上讲，插值过程有两个阶段：(1)找一个插值函数满足已知数据点，(2)对目标点进行插值函数评估。

然而在实际应用中，这个二阶段法并不总是最好的办法。此法与在需要每次由 $N$ 个给

定值直接构造估计函数$f(x)$的方法相比，它在计算上典型地缺乏高效性，而且给人误差问题也值得怀疑。大多数实用方法是，由一个邻近点$f(x_i)$开始，考虑其它相关的$f(x_i)$信息，然后进行一连串的越来越小的修正(希望如此)，该过程的运算次数一般为$O(N^2)$。如果进行得顺利，最后一次的修正规模最小，而且该修正值可作为误差的一个非正式的(然而并不是严格的)界限。

对多项式插值的情况，有时候确实对插值多项式的系数感兴趣，尽管用它们来估计插值函数也并不满意。我们将在第3.5节中来讨论这个问题。

局部插值，使用有限个"最近邻"点，给出插值函数$f(x)$。一般说来它没有连续的一阶导数或高阶导数。这是因为$x$取的是表中给出的值$x_i$，插值法所连接的这些表中的点都是"局部的"。(如果这样的连接允许发生任何地方，则在那一点插值函数本身将不连续，这是不好的做法!)

在考虑导数连续性要求时，就必须使用一种称为**样条**函数的"呆板"的**插值**方法了。样条是两个已知列表点之间的多项式函数，但它的系数却"有点"非局部确定的。这种非局部性是用来保证该插值函数的若干阶导数的整体平滑性。三次样条插值(第3.3节)是最为通用的方法。其产生的插值函数二阶导数连续。样条法比多项式法趋于稳定些，它在列表点之间摆动的概率要小些。

插值法中使用的节点数目(减1)称为该**插值的阶**。尤其对多项式插值，增加阶数并不一定能提高精度。如果添加的点离感兴趣的点$x$相差太远，则造成带有额外限制点的高阶多项式，会在列表值之间产生振荡现象。这种振荡可能与"真实"函数的形状毫无关系(图3.0.1)。自然，增加所求节点附近节点的数目，通常会有所帮助。但一点点改进意味着要增加大量表中的数值，故并不总是可取的。

除非有可靠的证据证明，插值函数的形式接近于真实函数f，那么使用高阶插值是个好主意。我们推荐使用3至4个节点的插值，5至6个节点也是可以容忍的。除非对估计的误差控制得相当严格，一般我们极少使用再多的节点了。

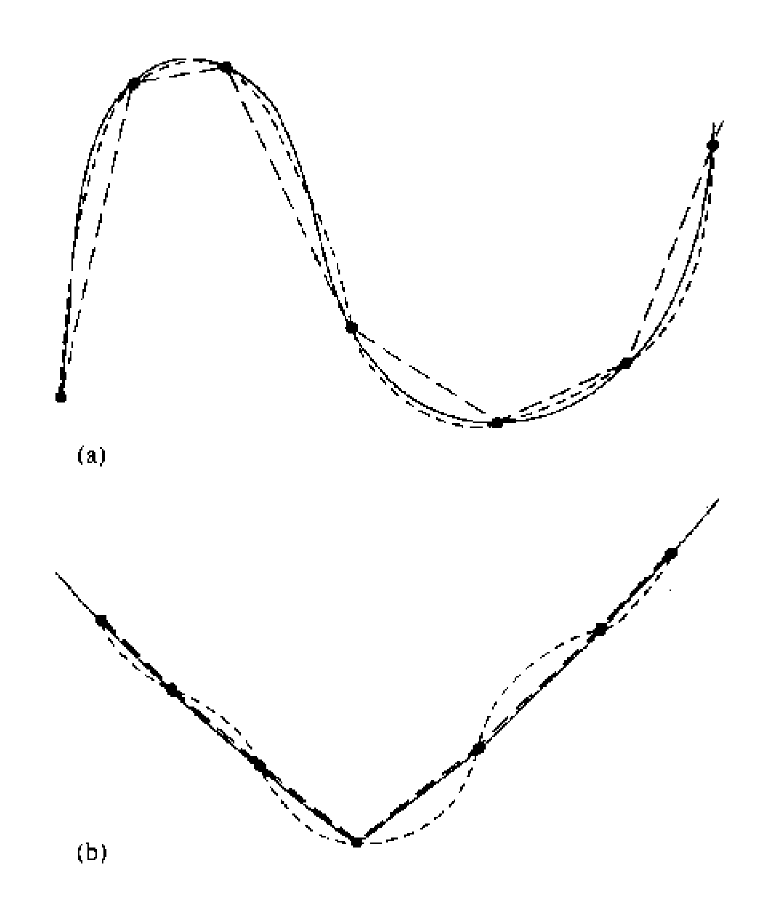
如果数据表中包含的点多于插值所需的数目，则在每次插值时，就必须选取表中合适位置的点。尽管这项工作严格说来，并不是插值法专题的一部分，但却非常重要(经常做得很不好)。我们将在第3.4节中讨论。

给出的内插法程序也是外推法程序。它在第十六章中的一个重要应用是，常微分方程的积分解法。那里要引起特别注意的是误差控制。否则，外推法的不可靠性就要过分强调：内插函数用作外推函数，当参数$x$在已知区间之外，且其距离大于表中点的典型间距时，所得结果与真实值会相差甚远。

插值可以是多维的，如函数$f(x,y,z)$。一般情况下，多维插值伴随着一系列一维插值。这个问题在第3.6节中讨论。

## 3.1 多项式内插和外推

通过任意两点只有唯一一条直线，通过任意三点只有唯一一个平面，等等。通过$N$个点$y_1=f(x_1)$，$y_2=f(x_2)$，…，$y_N=f(x_N)$的$N-1$维内插多项式，可用拉格朗日(Lagrange)经典公式明确地表示出来。



(a)平滑函数(实线),用高阶多项式插值表示(图示的虚线),比用低级多项式插值表示(图中一段段的折线)要更精确些。(b)具有拐点很陡或高阶导数变化迅速的函数,用高阶多项式近似(虚线),由于太"呆板",比用低阶多项式近似(折线)精度要差一些。即使象指数函数或有理函数这样的平滑函数,用高阶多项式近似也是很糟糕的。

图3.0.1

$$P(x)=\frac{(x-x_2)(x-x_3)\cdots(x-x_N)}{(x_1-x_2)(x_1-x_3)\cdots(x_1-x_N)}y_1+\frac{(x-x_1)(x-x_3)\cdots(x-x_N)}{(x_2-x_1)(x_2-x_3)\cdots(x_2-x_N)}y_2$$
$$+\cdots+\frac{(x-x_1)(x-x_2)\cdots(x-x_{N-1})}{(x_N-x_1)(x_N-x_2)\cdots(x_N-x_{N-1})}y_N \tag{3.1.1}$$

其中共有 $N$ 个项,每个项都是一个 $N-1$维多项式。对每一项除了一个 $x_i$ 使其值为 $y_i$ 外,对其它所有 $x_i$ 值均为零。

直接使用拉格朗日公式并无什么严重的错误,但也不是非常好的办法。这种求解算法没有给出误差估计,并且不适合编程。一个较好的算法(构造同样的唯一的插值多项式)是尼维尔(Neville)算法,它与埃特金(Aitken)算法有关联,有时甚至容易弄混,但后者现在看来已经过时不用了。

令 $P_1$ 是经过点$(x_1,y_1)$的 $x$ 零阶多项式的值(即常数),故 $P_1=y_1$。类似的方法定义 $P_2$,$P_3$,…,$P_N$,再令 $P_{12}$为过点$(x_1,y_1)$和$(x_2,y_2)$的 $x$ 一阶多项式的值;类似方法定义 $P_{23}$,$P_{34}$,…,$P_{(N-1)N}$。对高阶多项式做类似定义,直至 $P_{123\cdots N}$,它是对过所有 $N$ 个点的插值多项式的值,即所求的解。各个不同的 $P$ 形成一张"表",左边为"祖先",一直推到最右边的一个"后

代”。比如，当 $N=4$ 时，

$$
\begin{array}{llll}
x_1: & y_1 = P_1 & & \\
 & & P_{12} & \\
x_2: & y_2 = P_2 & & P_{123} \\
 & & P_{23} & \qquad P_{1234} \\
x_3: & y_3 = P_3 & & P_{234} \\
 & & P_{34} & \\
x_4: & y_4 = P_4 & &
\end{array}
\tag{3.1.2}
$$

尼维尔算法用递推的方法填写表中的数，每次一列，从左到右。它基于“子女”$P$ 和其两个“双亲”之间的关系，得

$$P_{i(i+1)\cdots(i+m)} = \frac{(x - x_{i+m})P_{i(i+1)\cdots(i+m-1)} + (x_i - x)P_{(i+1)(i+2)\cdots(i+m)}}{x_i - x_{i+m}} \tag{3.1.3}$$

因为两个双亲值已由点 $x_i \cdots x_{i+m-1}$ 得到，故该递推式可求。

对递推式(3.1.3)的一个改进是，记下双亲值与子女值之间的微小“差值”，即定义（对 $m=1,2,\cdots N-1$）

$$
\begin{aligned}
C_{m,i} &\equiv P_{i\ldots(i+m)} - P_{i\ldots(i+m-1)} \\
D_{m,i} &\equiv P_{i\ldots(i+m)} - P_{(i+1)\ldots(i+m)}
\end{aligned}
\tag{3.1.4}
$$

则由式(3.1.2)很容易推出关系式：

$$
\begin{aligned}
D_{m+1,i} &= \frac{(x_{i+m+1} - x)(C_{m,i+1} - D_{m,i})}{x_i - x_{i+m+1}} \\
C_{m+1,i} &= \frac{(x_i - x)(C_{m,i+1} - D_{m,i})}{x_i - x_{i+m+1}}
\end{aligned}
\tag{3.1.5}
$$

对每一层 $m$，修正后的 $C$ 和 $D$ 都使插值高一级。最后的解 $P_{1\ldots N}$ 等于任意 $y_i$ 加上一些 $C$ 和(或)$D$ 之和，这些 $C$ 和 $D$ 形成一条穿过家族树到达最右端子女的路径。

下面是一个有 $N$ 个输入点多项式内插或外推程序。注意输入数组设为单位偏移量，若数组为零偏移量，记住要减1(参看第1.2节)：

```
#include <math.h>
#include "nrutil.h"

void polint(float xa[], float ya[], int n, float x, float *y, float *dy)
  给定数组 xa[1..n],ya[1..n]及数值 x,本程序返回数值 y 及误差估计 dy。如果 P(x)为使 P(xai)=yai,i=1,…,n 的
  n－1阶多项式，则返回值 y=P(x)。
{
    int i,m,ns=1;
    float den,dif,dift,ho,hp,w;
    float *c,*d;

    dif=fabs(x-xa[1]);
    c=vector(1,n);
    d=vector(1,n);
    for (i=1;i<=n;i++) {                这里用 ns 作为表中最近入口的指针
        if ((dift=fabs(x-xa[i])) < dif) {
            ns=i;
            dif=dift;
        }
        c[i]=ya[i];                      初始化表 c 和表 d
        d[i]=ya[i];
```

```
*y=ya[ns--];                                              y的初始估计
for (m=1;m<n;m++) {                                       对表中的每一列
    for (i=1;i<=n-m;i++) {                                对当前c和d进行循环,并更新它们的值
        ho=xa[i]-x;
        hp=xa[i+m]-x;
        w=c[i+1]-d[i];
        if ( (den=ho-hp) == 0.0) nrerror("Error in routine polint");   该错误仅当两个输入的xa
                                                          值(在误差范围内)相同时
        den=w/den;                                        才发生
        d[i]=hp*den;                                      c和d值在此处被更新
        c[i]=ho*den;
    }
    *y += (*dy=(2*ns < (n-m) ? c[ns+1] : d[ns--]));
    表中的一列处理完后,判断要把修正值c还是d加到y的累积值上,就是说,选择表中的哪一条路径——是上
    边的分叉还是下边的。我们的做法是走表中最"直"的路径,并更新ns值以记录所走的路径。本程序中所得y是在
    点x上的有偏估计值,最后用dy表示误差
}
free_vector(d,1,n);
free_vector(c,1,n);
}
```

常常需要调用程序 **polint**,而调用时用实际具有偏移量的数组替换虚设变量 xa 和 ya。例如,构造 **polint**(&xx[14],&yy[14],4,x,y,dy)程序,使它在列表值 xx[15..18]、yy[15..18]上实现4点内插。

## 3.2 有理函数内插法和外推法

有些函数用多项式近似效果不好,但用有理函数,即多项式的商来估计效果却很好。我们用 $R_{i(i+1)\cdots(i+m)}$ 表示过 $m+1$ 个点 $(x_i,y_i)\cdots(x_{i+m},y_{i+m})$ 的有理函数。为更清楚起见,假定

$$R_{i(i+1)\cdots(i+m)}=\frac{P_\mu(x)}{Q_\nu(x)}=\frac{p_0+p_1x+\cdots p_\mu x^\mu}{q_0+q_1x+\cdots+q_\nu x^\nu}\tag{3.2.1}$$

因为有 $\mu+\nu+1$ 个未知的 $p$ 和 $q$($q_0$ 任意),则有

$$m+1=\mu+\nu+1\tag{3.2.2}$$

在把插值函数定义为有理函数时,必须给出分子和分母多项式的阶数。

有理函数比多项式函数优越,粗略而言,是因为它们能够模拟具有极点的函数。极点是指等式(3.2.1)中分母的零点。如果要插值的函数本身有极点的话,则在实数 $x$ 处就有可能发生这些极点。更为常见的情况是,函数 $f(x)$ 对所有有限实数 $x$ 都是有穷的,但在 $x$ 复平面内存在极点,是解析连续的。即使限定为实数 $x$,这些极点也会使多项式逼近彻底失败,正如它们会使 $x$ 的无穷级数达不到收敛一样。如果在复平面内,沿列表中的 $m$ 个点画一个圆,除非最近的极点也远离此圆,否则就别指望多项式插值结果很好。相比而言,只要有理函数的分母中,$x$ 有足够高的幂次以避免(消除)近处的极点,有理函数逼近的效果还不错。

对插值问题,构造有理函数就是为了使其能够通过一组选定的列表函数值。然而应该提及的是,有理函数逼近可以用于解析的工作中。有人曾构造了一个有理函数逼近,其准则为:等式(3.2.1)的有理函数本身的幂级数展开,应与欲求函数 $f(x)$ 的幂级数展开前 $m+1$ 项一致。这被称为**帕得**(Padé)逼近,这将在第5.12节中讨论。

Bulirsch 和 Stoer 发现了一种尼维尔型的算法，可以对表中的数据进行有理函数外推。按列构造类似等式(3.1.2)的表，进而导出最后的结果和误差估计。Bulirsch-Stoer 算法产生一个称作对角的有理函数，它分子和分母阶数相等(若 $m$ 为偶数)，或者分母的阶数比分子的阶数高一阶(若 m 为奇数，参见上面的等式(3.2.2))。该算法的推导参见[1]。类似于多项式逼近的等式(3.1.3)，该算法也可用递推关系精确地总结为：

$$R_{i(i+1)\cdots(i+m)} = R_{(i+1)\cdots(i+m)} + \frac{R_{(i+1)\cdots(i+m)} - R_{i\cdots(i+m-1)}}{\left(\frac{x-x_i}{x-x_{i+m}}\right)\left(1-\frac{R_{(i+1)\cdots(i+m)} - R_{i\cdots(i+m-1)}}{R_{(i+1)\cdots(i+m)} - R_{(i+1)\cdots(i+m-1)}}\right)-1} \tag{3.2.3}$$

该递推式表示，可由过 $m$ 个点的有理函数和过 $m-1$ 个点的有理函数(等式(3.2.3)中 $R_{(i+1)\cdots(i+m-1)}$)导出过 $m+1$ 个点的有理函数。初始值为

$$R_i = y_i \tag{3.2.4}$$

及

$$R \equiv [R_{i(i+1)\cdots(i+m)} \quad 当\ m = -1\ 时] = 0 \tag{3.2.5}$$

现在，跟上面等式(3.1.4)和(3.1.5)完全一样，将递推式(3.2.3)转化为仅含细小差别的式子

$$\begin{aligned} C_{m,i} &\equiv R_{i\cdots(i+m)} - R_{i\cdots(i+m-1)} \\ D_{m,i} &\equiv R_{i\cdots(i+m)} - R_{(i+1)\cdots(i+m)} \end{aligned} \tag{3.2.6}$$

注意它们满足关系式

$$C_{m+1,i} - D_{m+1,i} = C_{m,i+1} - D_{m,i} \tag{3.2.7}$$

该式在证明下面的递推式时是非常有用的，

$$\begin{aligned} D_{m+1,i} &= \frac{C_{m,i+1}(C_{m,i+1} - D_{m,i})}{\left(\frac{x-x_i}{x-x_{i+m+1}}\right)D_{m,i} - C_{m,i+1}} \\ C_{m+1,i} &= \frac{\left(\frac{x-x_i}{x-x_{i+m+1}}\right)D_{m,i}(C_{m,i+1} - D_{m,i})}{\left(\frac{x-x_i}{x-x_{i+m+1}}\right)D_{m,i} - C_{m,i+1}} \end{aligned} \tag{3.2.8}$$

该递推关系由下面的函数来实现，其用法与第3.1节中的 **polint** 程序完全相同。再次提醒，输入数组假定为单位偏移量(参见第1.2节)。

```
#include <math.h>
#include "nrutil.h"
#define TINY 1.0e-25                         一个小的数
#define FREERETURN {free_vector(d,1,n);free_vector(c,1,n);return;}

void ratint(float xa[], float ya[], int n, float x, float *y, float *dy)
    给定数组 xa[1..n], ya[1..n]及数值 x，本程序返回 y 的值及精度估计 dy。返回值是通过几个点(xa_i,ya_i)，i=1...n
    的有理函数在 x 处的值。
{
    int m,i,ns=1;
    float w,t,hh,h,dd,*c,*d;

    c=vector(1,n);
    d=vector(1,n);
    hh=fabs(x-xa[1]);
```

```
    for (i=1;i<=n;i++) {
        h=fabs(x-xa[i]);
        if (h == 0.0) {
            *y=ya[i];
            *dy=0.0;
            FREERETURN
        } else if (h < hh) {
            ns=i;
            hh=h;
        }
        c[i]=ya[i];
        d[i]=ya[i]+TINY;                 须用 TINY 来防止少见的0除0的情况
    }
    *y=ya[ns--];
    for (m=1;m<n;m++) {
        for (i=1;i<=n-m;i++) {
            w=c[i+1]-d[i];
            h=xa[i+m]-x;                 因为在初始循环中已测试过，这里 h 不会为零
            t=(xa[i]-x)*d[i]/h;
            dd=t-c[i+1];
            if (dd == 0.0) nrerror("Error in routine ratint");   这种错误情况
            dd=w/dd;                                             表明该插值函数在欲
            d[i]=c[i+1]*dd;                                      求的 x 处存在极点
            c[i]=t*dd;
        }
        *y += (*dy=(2*ns < (n-m) ? c[ns+1] : d[ns--]));
    }
    FREERETURN
}
```

**参考文献和进一步读物：**


Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §2.2. [1]

Cuyt, A., and Wuytack, L. 1987, *Nonlinear Methods in Numerical Analysis* (Amsterdam: North-Holland), Chapter 3.


## 3.3 三次样条插值

给定一个列表显示的函数 $y_i=y(x_i)$，$i=1\ldots N$，把注意力放在 $x_j$ 和 $x_{j+1}$ 之间的一个特殊的区间上。该区间的线性插值公式为

$$y = Ay_j + By_{j+1} \tag{3.3.1}$$

其中

$$A \equiv \frac{x_{j+1}-x}{x_{j+1}-x_j} \qquad B \equiv 1 - A = \frac{x-x_j}{x_{j+1}-x_j} \tag{3.3.2}$$

等式(3.3.1)和(3.3.2)是拉格朗日插值公式(3.1.1)的特殊情况。

因为它是(分段)线性的，等式(3.3.1)在每一区间内的二阶导数为零，在横座标为 $x_j$ 处的二阶导数不定义或无限。三次样条插值的目的就是要得到一个内插公式，不论在区间内或其边界上，其一阶导数平滑，二阶导数连续。

做一个与事实相反的假设，除 $y_i$ 的列表值之外，我们还有函数二阶导数 $y''$ 的列表值，即一系列的 $y_i''$ 值。则在每个区间内，我们可以在等式(3.3.1)的右边加上一个三次多项式，

其二阶导数从左边的 $y_j''$ 值线性变化到右边的 $y_{j+1}''$ 值。这么做，我们便得到了所需的连续的二阶导数。如果我们还将该三次多项式构成在 $x_j$ 和 $x_{j+1}$ 处为零，这样就不会破坏在终点 $x_j$ 和 $x_{j+1}$ 处与列表函数值 $y_j$ 和 $y_{j+1}$ 的一致性。

做一些辅助计算便可知，仅有一种办法才能进行这种构造，即用

$$y = Ay_j + By_{j+1} + Cy_j'' + Dy_{j+1}'' \tag{3.3.3}$$

来代替(3.3.1)，其中 A 和 B 由(3.3.2)定义，且

$$C \equiv \frac{1}{6}(A^3 - A)(x_{j+1} - x_j)^2$$
$$D \equiv \frac{1}{6}(B^3 - B)(x_{j+1} - x_j)^2 \tag{3.3.4}$$

注意，等式(3.3.3)和(3.3.4)对自变量 $x$ 的依赖，是完全通过 $A$ 和 $B$ 对 $x$ 的线性依赖，以及 $C$ 和 $D$(通过 $A$ 和 $B$)对 $x$ 的三次依赖。

我们可以很容易地验证，$y''$ 事实上是该插值多项式的二阶导数。使用 $A,B,C,D$ 的定义对 $x$ 求等式(3.3.3)的导数，计算 $dA/dx$，$dB/dx$，$dC/dx$ 及 $dD/dx$。结果为：一阶导数

$$\frac{dy}{dx} = \frac{y_{j+1} - y_j}{x_{j+1} - x_j} - \frac{3A^2 - 1}{6}(x_{j+1} - x_j)y_j'' + \frac{3B^2 - 1}{6}(x_{j+1} - x_j)y_{j+1}'' \tag{3.3.5}$$

二阶导数

$$\frac{d^2y}{dx^2} = Ay_j'' + By_{j+1}'' \tag{3.3.6}$$

因为 $x=x_j$ 时 $A=1$，$x=x_{j+1}$ 时 $A=0$，而 $B$ 正相反，则式(3.3.6)表明 $y''$ 恰为列表函数的二阶导数。而且该二阶导数在两个开区间 $(x_{j-1},x_j)$ 和 $(x_j,x_{j+1})$ 上是连续的。

现在唯一的问题是，我们假设 $y_i''$ 是已知的，而实际上并不知道。然而，我们仍不要求从等式(3.3.5)算出的一阶导数，在两个区间的边界处是连续的。三次样条的关键思想，就是要求这种连续性并用它求得等式的二阶导数 $y_i''$。

设等式(3.3.5)在区间 $(x_{j-1},x_j)$ 上，对 $x=x_j$ 求得的值，等于同一等式在区间 $(x_j,x_{j+1})$ 上，对 $x=x_j$ 求得的值，便可得到所求方程。重新整理得到(对 $j=2,\cdots,N-1$)

$$\frac{x_j - x_{j-1}}{6}y_{j-1}'' + \frac{x_{j+1} - x_{j-1}}{3}y_j'' + \frac{x_{j+1} - x_j}{6}y_{j+1}'' = \frac{y_{j+1} - y_j}{x_{j+1} - x_j} - \frac{y_j - y_{j-1}}{x_j - x_{j-1}} \tag{3.3.7}$$

有 $N-2$ 个线性方程，但却有 $N$ 个未知数 $y_i''$，$i=1,\ldots,N$，因此，有一个具有两个参数的可能解集。

为求得唯一解，需要给出两个进一步的条件，一般取 $x_1$ 和 $x_N$ 处的边界条件。最常见的做法有

- 设 $y_1''$ 和 $y_N''$ 之一或两个都为零，得到所谓的**自然三次样条**函数，其一个或两个边界的二次导数为零，或者
- 设 $y_1''$ 或 $y_N''$ 为等式(3.3.5)计算得到的值，使得该插值函数的一阶导数在一个或两个边界处有特定的值。

三次样条插值特别实用的原因之一，在于有两个附加边界条件的方程组(3.3.7)，它不仅是线性的，而且也是三对角的。每个 $y_j''$ 仅与其最近邻的 $j\pm1$ 的值有关。因此，方程可以用三对角算法(第2.4节)在 $O(N)$ 次运算内求解。该算法非常简明，很容易正确地构造出样条

计算的程序。但是这使得程序不象式(3.3.7)的实现那样完全透明，所以我们鼓励读者将下面程序与程序 **tridag**(第2.4节)比较一下，仔细地研究研究。数组假定是单位偏移量，如果是零偏移量，参见第1.2节。

```
#include "nrutil.h"

void spline(float x[], float y[], int n, float yp1, float ypn, float y2[])
    给出数组x[1..n]和y[1..n]构成一个列表函数，即y_i=f(x_i)，其中x_1<x_2<…<x_N，还有插值函数在第1个和第
    n个点处的一阶导数值yp1和ypn，相应地，本程序返回数组y2[1..n]，其值为插值函数在表中x_i点的二阶导数。如
    果yp1和(或)ypn大于或等于1×10^30，则本程序将相应的边界条件设为自然样条，即边界的二阶导数为零。
{
    int i,k;
    float p,qn,sig,un,*u;

    u=vector(1,n-1);
    if (yp1 > 0.99e30)                下边界条件设成"自然的"
        y2[1]=u[1]=0.0;
    else {                            否则有特定的一阶导数
        y2[1] = -0.5;
        u[1]=(3.0/(x[2]-x[1]))*((y[2]-y[1])/(x[2]-x[1])-yp1);
    }
    for (i=2;i<=n-1;i++) {            三对角算法的分解循环
        sig=(x[i]-x[i-1])/(x[i+1]-x[i-1]);         y2和u为被分解因子的临时变量
        p=sig*y2[i-1]+2.0;
        y2[i]=(sig-1.0)/p;
        u[i]=(y[i+1]-y[i])/(x[i+1]-x[i]) - (y[i]-y[i-1])/(x[i]-x[i-1]);
        u[i]=(6.0*u[i]/(x[i+1]-x[i-1])-sig*u[i-1])/p;
    }
    if (ypn > 0.99e30)                上边界条件设成"自然的"
        qn=un=0.0;
    else {                            否则有特定的一阶导数
        qn=0.5;
        un=(3.0/(x[n]-x[n-1]))*(ypn-(y[n]-y[n-1])/(x[n]-x[n-1]));
    }
    y2[n]=(un-qn*u[n-1])/(qn*y2[n-1]+1.0);
    for (k=n-1;k>=1;k--)              三对角算法的回代循环
        y2[k]=y2[k]*y2[k+1]+u[k];
    free_vector(u,1,n-1);
}
```

重要的是要记住，在处理数组 $x_i$ 和 $y_i$ 构成的列表函数时，程序 **spline** 仅能调用一次。一旦调用过后，对任意的 $x$，插值函数的值通过调用(要多少次就调多少次)一个独立的程序 **splint**("*spline interpolation*"的字头缩写)来得到。

```
void splint(float xa[], float ya[], float y2a[], int n, float x, float *y)
    给出两个数组xa[1..n]和ya[1..n]作为列表函数(按xa_i的顺序)，数组y2a[1..n]由上面spline程序的输出给定，
    并给定x的值，本程序返回x的三次样条插值结果y。
{
    void nrerror(char error_text[]);
    int klo,khi,k;
    float h,b,a;

    klo=1;                            用二分法查找表的正确位置。如果用随机的x值连续调用本函
    khi=n;                            数，这么做是最佳的。如果连续调用时，x是有序的且间距不大，最
    while (khi-klo > 1) {             好把前一次调用时klo和khi的值保存起来，在下次调用
        k=(khi+klo) >> 1;             时先检测一下它们的值是否还合适。
        if (xa[k] > x) khi=k;
```

```
        else klo=k;
    }                                      klo 和 khi 为输入 x 的上下界
    h=xa[khi]-xa[klo];
    if (h == 0.0) nrerror("Bad xa input to routine splint");          xa 的值不能相同
    a=(xa[khi]-x)/h;
    b=(x-xa[klo])/h;                       三次样条多项式求值
    *y=a*ya[klo]+b*ya[khi]+((a*a*a-a)*y2a[klo]+(b*b*b-b)*y2a[khi])*(h*h)/6.0;
}
```

## 3.4 如何搜索有序表

假定决定用一些特殊的插值方法，如用四阶多项式插值，根据表中 $x_i$ 及 $f_i$ 的值计算出函数 $f(x)$。那么，对给出的特定值 $x$，要求计算其函数值，就需要一个快速的方法找到在表中 $x_i$ 的位置。这个问题如归于数值分析不一定合适，但实际中却很容易碰到，所以不能忽略它。

这个问题的标准描述为：已知横坐标数组 xx[j]，j=1，2，…，n，按递增或递减的顺序排列，对给出的数 x，求整数 j，使得 x 在 xx[j]与 xx[j+1]之间。为了查找方便，首先定义两个附加的数组元素 xx[0]和 xx[n+1]，分别代表正负无穷大（这样仍然保持表中元素的单调性）。则 j 总是在0到 n 之间了，即可用0作为表的一端界限，n 作为另一端界限。

在大多数情况下，当上面所述的要求满足时，再没有比**二分法**更好的办法了，它能在大约 $\log_2 n$ 次试验内找到表中正确的位置。在上一节的样条法求值程序 **splint** 中，我们已经使用了二分法，所以可以参考一下。另外，二分法程序一般可如下构造：

```
void locate(float xx[], unsinged long n, float x, unsigned long *j)
    已知数组 xx[1..n]及 x 的值，返回 j 的值，使得 x 在 xx[j]和 xx[j+1]之间。xx 必须是单调的，递增或递减均可。若
    返回 j=0或 j=n 表明 x 越界了。
{
    unsigned long ju,jm,jl;
    int ascnd;

    jl=0;                                      初始化上下边界
    ju=n+1;
    ascnd=(xx[n] > xx[1]);
    while (ju-jl > 1){                         如果还没有做完，则计算中点
        jm=(ju+jl) >> 1;
        if (x > xx[jm] == ascnd)
            jl=jm;                             替换下界
        else
            ju=jm;                             否则应替换上界
    }                                          循环至条件满足
    *j=jl;                                     设置输出量并返回
}
```

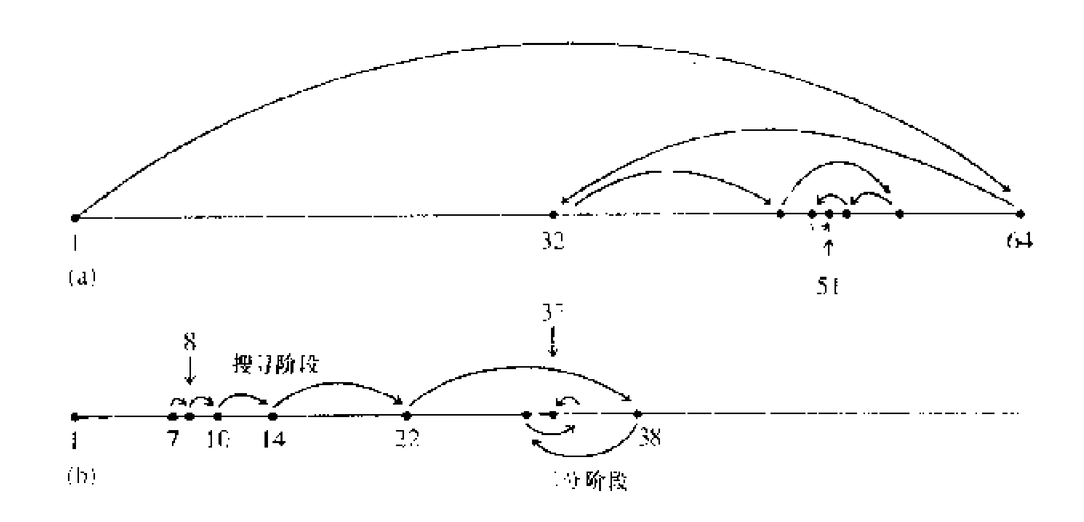
数组 xx 假定是单位偏移量。如果是零偏移数组，请记住要将 xx 的地址减 1，而且返回的值 j 也要减 1。

### 3.4.1 用相关数值进行搜索

有时会遇到多次查表的情况，而且是用几乎相同的横坐标进行连续查询。例如，可能正在生成一个用在微分方程左端的函数；如像我们将在第十六章中看到的那样，大部分微分方

程的积分要在向前或向后跳移的一些点上调用有端查询等，但是，在此积分方向上，它的移动趋势很慢。

在这种情况下，每次调用都从头开始做完全二分处理是很浪费的。下面的程序，不是从头而是从表中某个预测的位置开始查表。它的第一次搜寻，不论递增还是递减，都是要步长为1，下一次为2，再一次为4，等等，直到要查的值落在该区间内为止。然后，在所得的区间内进行二分处理。在最坏的情况下，本程序大概要比上面的 **locate** 程序慢两倍（如果搜寻的范围遍及整张表）。在最好的情况下，即所求点通常离前值很近的情况，要比 **locate** 程序快 $\log_2 n$ 倍。图3.4.1是对这两个程序的比较。




(a)程序 **locate** 用二分从头查找有序表。这里显示的是在长度为64的表中，收敛到元素51的过程。(b)程序 **hunt** 是从表中前一次的位置开始，按递增步长搜寻，然后用二分法收敛。这里显示的是一个令人非常不满意的例子，从元素7开始收敛到元素37。比较满意的例子应是查找邻近的元素，比如8，仅需要二次查找。

图3.4.1

```
void hunt(float xx[], unsigned long n, float x, unsigned long *jlo)
    已知数组xx[1..n]及数值x，返回jlo的值，使得x在xx[jlo]与xx[jlo+1]之间。xx必须是单调的，或递增，或递减。
    若返回jlo=0或jlo=n则表明x越界。输入时jlo的值是作为其输出值的预测值
{
    unsigned long jm,jhi,inc;
    int ascnd;

    ascnd=(xx[n] > xx[1]);                递增的顺序为真，否则为假
    if (*jlo <= 0 || *jlo > n) {          预测值不准，无参考价值，直接转二分处理
        *jlo=0;
        jhi=n+1;
    } else {
        inc=1;                            设置搜索步长
        if (x >= xx[*jlo] == ascnd) {     向上搜索
            if (*jlo == n) return;
            jhi=(*jlo)+1;
            while (x >= xx[jhi] == ascnd) {       搜索未结束
                *jlo=jhi;
                inc += inc;               增量加倍
                jhi=(*jlo)+inc;
```

```
            if (jhi > n) {          搜寻结束，已至表尾
                jhi=n+1;
                break;
            }
        }                           再试
    } else {                        搜寻结束，值区间找到
        if (*jlo == 1) {            向下搜索
            *jlo=0;
            return;
        }
        jhi=(*jlo)--;
        while (x < xx[*jlo] == ascnd) {     搜寻未结束
            jhi=(*jlo);
            inc <<= 1;              加倍增量
            if (inc >= jhi) {       搜寻结束，已至表尾
                *jlo=0;
                break;
            }
            else *jlo=jhi-inc;
        }                           再试
    }                               搜寻结束，值区间找到
}                                   搜寻完成，开始进入最后的二分阶段
while (jhi-(*jlo) != 1) {
    jm=(jhi+(*jlo)) >> 1;
    if (x > xx[jm] == ascnd)
        *jlo=jm;
    else
        jhi=jm;
}
}
```

如果数组 xx 是零偏移量，参考上面 **locate** 程序后的注释。

### 3.4.2　写在 Hunt 之后

问题：程序 **locate** 和 **hunt** 返回索引指针 j，使求的值处在表项 xx[j]与 xx[j+1]之间，其中 xx[1..n]为表的总长度，但是使用象 **polint**(第3.1节)或 **ratint**(第3.2节)这样的程序，是为了要想得到 m 点插值，就必须提供长度为 m 的较短的数组 xx 和 yy。怎么办呢？

解答：计算

k=IMIN(IMAX(j-(m-1)/2,1),n+1-m)

(宏调用 IMIN 和 IMAX 是求两个整型参数的最小值和最大值，参见第1.2节及附录B)。该表达式产生的是，m 点集合中最左边元素的索引指针，这 m 个点的中心在 j 和 j+1之间(在可能范围内)，但左边界为1，右边界为 n。C 语言允许用数组的地址偏移量 k 来调用插值程序，例如，

Polint (&xx[k-1], &yy[k-1],m,…)

## 3.5　插值多项式的系数

有些时候想要知道的，并不是经过少数几个点的插值多项式的函数值，而是该多项式的系数。这些系数的实际用处可以是，例如同时计算函数的插值和它的几个导数(参见第5.3节)，或者求列表函数段与某个其它函数的卷积，这个其它函数的矩(即它与 $x$ 的幂函数卷积)是解析已知的。

然而，一定要肯定这些系数确为所需的。一般说来，插值多项式的系数的确定，不如在特定点求函数值那么精确。因此，仅仅为计算插值而确定系数并不是个好办法。那样计算得到的数值并不能精确地通过列表中的点，而用第3.1节—第3.3节中的程序所计算的值，能精确地通过这些列表点。

读者也不应该将插值多项式(及其系数)错当成它的类似问题：经过一系列数据点进行多项式的**最佳拟合**。拟合是个求**平滑**函数的过程，因为拟合的系数个数一般要远比数据点的数目少。所以，即使在列表值出现统计误差时，拟合的系数也可以精确而稳定地确定出来(见第14.8节)。而对于系数个数等于列表点个数的插值法，是把所有列表中的数据都当成准确无误的。如果其中含有统计误差，则可能导致列表点之间产生插值多项式的摆动。

跟以前一样，令列表中的点为 $y_i \equiv y(x_i)$。如果插值多项式写成

$$y = c_0 + c_1 x + c_2 x^2 + \cdots + c_N x^N \tag{3.5.1}$$

则 $c_i$ 应满足线性方程

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^N \\ 1 & x_1 & x_1^2 & \ldots & x_1^N \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_N & x_N^2 & \ldots & x_N^N \end{bmatrix} \cdot \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_N \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_N \end{bmatrix} \tag{3.5.2}$$

这是一个**范德蒙矩阵**，见第2.8节所述。原则上可以用解线性方程的标准技术(第2.3节)来求解方程(3.5.2)。然而，用第2.8节推导出来的特殊方法对大的维数 $N$ 更为有效，因此这方法要好一些。

记住，范德蒙问题可能是相当病态的。在这种情况下，找不到能给出非常精确解的数值方法。但这些情况并不意味着，采用第3.1节中的方法计算插值会有什么困难，只是在计算系数时才会有困难。

和第2.8节中的程序一样，下面的程序也是由 G. Rybicki 提供的。注意程序中数组都假定为零偏移量。

```
#include "nrutil.h"

void polcoe(float x[], float y[], int n, float cof[])
    给定数组 x[0..n]及 y[0..n]表示列表函数 yi=f(xi)，本程序返回系数数组 cof[0..n]，使得 yi=∑j cofj xi^j
{
    int k,j,i;
    float phi,ff,b,*s;

    s=vector(0,n);
    for (i=0;i<=n;i++) s[i]=cof[i]=0.0;
    s[n] = -x[0];
    for (i=1;i<=n;i++) {              由递推求得主多项式P(x)的原系数s,
        for (j=n-i;j<=n-1;j++)
            s[j] -= x[i]*s[j+1];
        s[n] -= x[i];
    }
    for (j=0;j<=n;j++) {
        phi=n+1;
        for (k=n;k>=1;k--)                     量 phi = ∏j≠k(xj - xk)
            phi=k*s[k]+x[j]*phi;      作为的导数已找到P(xj)
        ff=y[j]/phi;
        b=1.0;                        拉格朗日公式中每一项的多项式系数由综合除法P(x)
```

```
        for (k=n;k>=0;k--) {                 除以(x - x_j)求得。解 c_k 为累计值
            cof[k] += b*ff;
            b=s[k]+x[j]*b;
        }
    }
    free_vector(s,0,n);
}
```

### 3.5.1 其它方法

还有一种技术是，采用已经介绍过的函数值的插值程序(第3.1节中的 **polint** 程序)。如果使用内插法(或外推法)求插值多项式在 $x=0$ 处的值，则此值显然为 $c_0$。现在可以将每个 $y_i$ 减去 $c_0$，再除以各自相应的 $x_i$。然后，去掉一个点(具有最小 $x_i$ 的点是很好的候选点)，再可以重复此过程求 $c_1$，如此继续。

这处理过程的稳定性并不能马上看出来，但我们发现一般情况下，它比直接处理的程序更要稳定些。这种方法的复杂度为 $N^3$，而处理一次为 $N^2$。然而，会发现对 $N$ 很大时这两种做法效果都不好，这是因为范德蒙问题本身是病态的。用单精度表示，$N$ 到8或10效果还满意；用双精度表示时，$N$ 值差不多可加倍。

```
#include <math.h>
#include "nrutil.h"

void polcof(float xa[], float ya[], int n, float cof[])
  给定数组 xa[0..n]及 ya[0..n]表示列表函数 ya_i=f(xa_i)，本程序返回系数数组 cof[0..n]，使得 ya_i = Σ_j cof_j xa_i^j。
{
    void polint(float xa[], float ya[], int n, float x, float *y, float *dy);
    int k,j,i;
    float xmin,dy,*x,*y;

    x=vector(0,n);
    y=vector(0,n);
    for (j=0;j<=n;j++) {
        x[j]=xa[j];
        y[j]=ya[j];
    }
    for (j=0;j<=n;j++) {
        polint(x-1,y-1,n+1-j,0.0,&cof[j],&dy);       由于程序 polint 使用的数组下标为[1..n]，
        xmin=1.0e38;                                  所以 x 和 y 的坐标要减去1。外推到 x=0
        k = -1;
        for (i=0;i<=n-j;i++) {                        求绝对值最小的余数 x_i
            if (fabs(x[i]) < xmin) {
                xmin=fabs(x[i]);
                k=i;
            }
            if (x[i]) y[i]=(y[i]-cof[j])/x[i];        同时化简所有的项
        }
        for (i=k+1;i<=n-j;i++) {                      并进行消去
            y[i-1]=y[i];
            x[i-1]=x[i];
        }
    }
    free_vector(y,0,n);
    free_vector(x,0,n);
}
```

如果点 $x=0$ 不在(或者至少不接近)列表 $x_i$ 值的范围,则插值多项式的系数一般来说将会变得非常大。因而,这些系数真正的“信息内容”就会和“平移引起”的大值有微小差别。这是一种病态的情况,会导致有效数字的丢失,并使求出的系数结果很差。这时,应当考虑重新定义原来的问题,把 $x=0$ 放到合适的地方。

另外一种比较糟的情况是,如果试图用太高阶的插值求平滑函数,则插值多项式自然会试图用高阶系数,即这些系数很大,但差不多相等而互相抵消了,仅靠最后几位小数来使列表值满足插值函数。这样做的效果,与插值多项式的值在其限定点之间摆动的内在倾向是一致的,即使机器的浮点精度非常非常好这种情况也会发生。上面的程序 **polcoe** 和 **polcof** 对这种异常情况都有不同程度的敏感。

这样,还能肯定这些系数是人们想求的吗?

## 3.6 二维或高维插值

对多维插值的情况,我们要根据一个列表值 $y$ 的 $n$ 维网格点及 $n$ 个一维向量,求估计函数 $y(x_1,x_2,\ldots,x_n)$,这 $n$ 个一维向量给出每组独立变量 $x_1,x_2,\ldots,x_n$ 的列表值。这里我们将不考虑非笛卡尔系统的插值问题,即不考虑表中函数值是 $n$ 维空间中的一些“随机”点,而不是直角坐标系中情况的。为清楚起见,我们仅考虑二维的情况,三维或高维的情况可以完全类推。

对二维情况,假定已知函数值的矩阵 ya[1..m][1..n]。另外还已知数组 x1a[1..m] 及数组 x2a[1..n]。这些输入量的关系由函数 $y(x_1,x_2)$ 确定

$$\text{ya[j][k]} = y(\text{x1a[j]},\text{x2a[k]}) \tag{3.6.1}$$

要求的是,用插值法计算函数 $y$ 在表中未列出点 $(x_1,x_2)$ 的值。

一个重要的概念是**方格网格点**,点 $(x_1,x_2)$ 落在其中,也就是说由列表中的四个点包围起了所求的内部点。为方便起见,我们从左下角起按反时针方向从1到4给这些点进行编号(见图3.6.1)。更精确地表示为,如果由

$$\begin{aligned}\text{x1a[j]} &\leqslant x_1 \leqslant \text{x1a[j+1]}\\ \text{x2a[k]} &\leqslant x_2 \leqslant \text{x2a[k+1]}\end{aligned} \tag{3.6.2}$$

定义 j 和 k,则

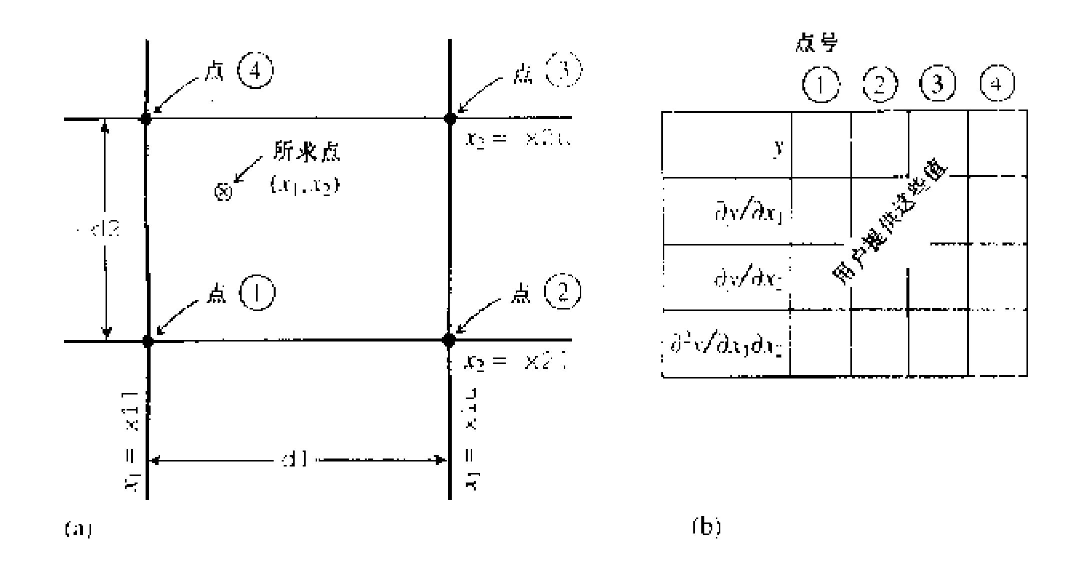
$$\begin{aligned}y_1 &\equiv \text{ya[j][k]}\\ y_2 &\equiv \text{ya[j+1][k]}\\ y_3 &\equiv \text{ya[j+1][k+1]}\\ y_4 &\equiv \text{ya[j][k+1]}\end{aligned} \tag{3.6.3}$$

最简单的二维插值是方格网格点上的二维**线性插值**。其公式为:

$$\begin{aligned}t &\equiv (x_1 - \text{x1a[j]})/(\text{x1a[j+1]} - \text{x1a[j]})\\ u &\equiv (x_2 - \text{x2a[k]})/(\text{x2a[k+1]} - \text{x2a[k]})\end{aligned} \tag{3.6.4}$$

(因此 $t$ 和 $u$ 都在0和1之间),

$$y(x_1,x_2) = (1-t)(1-u)y_1 + t(1-u)y_2 + tuy_3 + (1-t)uy_4 \tag{3.6.5}$$




(a)在二维插值程序 **bcuint** 和 **bcucof** 中对点进行标号,(b)对(a)中四个点的每个点,用户要提供下列函数值:两组一阶导数和一组混合导数,总共16个数值。

图3.6.1

二维线性插值经常很类似于"政府部门的行政工作"。在插值点从一个方格网格移到另一个方格网格时,插值函数的值是连续变化的。然而,插值函数的梯度在每个方格网格的边界上是不连续变化的。

将二维线性插值推广到高维时,有明显不同于原来方法的两个方面:可以使用高维插值获得插值函数精度的提高(是足够平滑的函数),而不必试图满足梯度和高阶导数的连续性。或者使用高维插值法在对方格网格边界点插值时,使一些导数更加平滑。我们相继考虑一下这两个方面。

## 3.6.1 用高维插值获得高精度

基本思想是,将高维插值问题分解成一系列一维插值问题。如果想在 $x_1$ 方向上做 m—1 维插值,在 $x_2$ 方向上做 n—1维插值,就首先设置一个 m×n 的由列表函数矩阵构成的子块,它包含了所求的点 $(x_1,x_2)$。接下来在 $x_2$ 方向,即子块的行方向上做 $m$ 次一维插值,求得在点(x1a[j],$x_2$),j=1,...,m 处的函数值。最后,在 $x_1$ 方向上做最后一次插值,便得到了解。下面的程序使用了第3.1节中的多项式插值程序 **polint**,及假定已设置好的子块(通过指针 float **ya 寻址,参看第1.2节):

```
#include "nrutil.h"

void polin2(float x1a[],float x2a[],float **ya,int m,int n,float x1,float x2,float *y,float *dy)
  给定数组 x1a[1..m]和 x2a[1..n]为独立变量,子矩阵 ya[1..m][1..n]表示由 x1a 和 x2a 定义的网格点的列表函数
  值;另外还知道独立变量 x1和 x2的值,本程序返回插值函数值 y 及精度表示 dy(仅根据 x1 方向上的插值)
{
    void polint(float xa[], float ya[], int n, float x, float *y, float *dy);
    int j;
```

```
float *ymtmp;

ymtmp=vector(1,m);
for (j=1;j<=m;j++) {                          行循环
    polint(x2a,ya[j],n,x2,&ymtmp[j],dy);      插值结果暂时存储起来
}
polint(x1a,ymtmp,m,x1,y,dy);                  最后一次插值
free_vector(ymtmp,1,m);
```

### 3.6.2 用高维插值获得高平滑度:双三次插值

我们给出两个不相关的通用方法。第一个方法通常称为**双三次插值**。

双三次插值要求使用者在每个坐标格点上不仅指定函数 $y(x_1,x_2)$ 的值，而且还要指定梯度 $\partial y/\partial x_1 = y_{,1}$，$\partial y/\partial x_2 = y_{,2}$ 及混合偏导数 $\partial^2 y/\partial x_1\partial x_2 = y_{,12}$。然后，便可在坐标刻度为 $t$ 和 $u$（等式3.6.4）处求得三次插值函数，它具有下述性质：(i)函数值及指定的导数都是由坐标格点重新精确产生的，(ii)当插值点由一个方格网格进入另一方格网格时，函数值及指定的导数值是连续变化的。

双三次插值方程并不需要正确说明多余的导数，理解这一点很重要！平滑的性质是由多重保证的，与指定导数的精度无关。如何获得这些指定的值，那是另一个单独的课题。获得的值越好，插值也就越精确。而且无论怎样插值都是平滑的。

最好是能知道导数的解析值，或是能够用数值方法精确地计算出方格点上的导数。次好的办法，是能够从列表中已有的方格点的函数值，通过数值差分来确定这些导数。有关的代码类似下面的形式（使用中心差分）：

```
y1a[j][k] =(ya[j+1][k]-ya[j-1][k])/(x1a[j+1]-x1a[j-1]);
y2a[j][k] =(ya[j][k+1]-ya[j][k-1])/(x2a[k+1]-x2a[k-1]);
y12a[j][k]=(ya[j+1][k+1]-ya[j+1][k-1])-ya[j-1][k+1]+ya[j-1][k-1])
          /((x1a[j+1]-x1a[j-1])*(x2a[k+1]-x2a[k-1]));
```

在方格网格的四个顶角点上，已知每个顶角的函数值 y 及导数 y1，y2，y12，构造方格网格内的双三次插值，其有两个步骤要做：首先，用下面的程序 **bcucof** 得到十六个量 $c_{ij}$，$i,j=1,\ldots,4$。（由函数值及导数值求 $c$ 的公式就是一个复杂的线性变换，其系数在求解过程中仅确定一次。可将其制成数表，以后便不用再考虑它了）。其次，将 $c$ 代换到下面任意几个或全部想要求的双三次函数值及导数的公式中：

$$
\begin{aligned}
y(x_1,x_2) &= \sum_{i=1}^{4}\sum_{j=1}^{4} c_{ij}t^{i-1}u^{j-1} \\
y_{,1}(x_1,x_2) &= \sum_{i=1}^{4}\sum_{j=1}^{4}(i-1)c_{ij}t^{i-2}u^{j-1} \\
y_{,2}(x_1,x_2) &= \sum_{i=1}^{4}\sum_{j=1}^{4}(j-1)c_{ij}t^{i-1}u^{j-2} \\
y_{,12}(x_1,x_2) &= \sum_{i=1}^{4}\sum_{j=1}^{4}(i-1)(j-1)c_{ij}t^{i-2}u^{j-2}
\end{aligned}
\qquad (3.6.6)
$$

其中 $t$ 和 $u$ 也由等式(3.6.4)给出。

```
void bcucof(float y[], float y1[], float y2[], float y12[], float d1, float d2, float **c)
```

给定数组y[1..4]、y1[1..4]、y2[1..4]及y12[1..4]，分别表示矩形坐标区域四个顶点(从左下角起按逆时针方向编号)的函数值、梯度及交叉导数，并且已知沿1,2方向上网格的单位长度d1和d2，本程序返回表c[1..4][1..4]，提供**bcuint**求双三次插值使用。

```
{
    static int wt[16][16]=
        { 1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,
        -3,0,0,3,0,0,0,0,-2,0,0,-1,0,0,0,0,
        2,0,0,-2,0,0,0,0,1,0,0,1,0,0,0,0,
        0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,
        0,0,0,0,-3,0,0,3,0,0,0,0,-2,0,0,-1,
        0,0,0,0,2,0,0,-2,0,0,0,0,1,0,0,1,
        -3,3,0,0,-2,-1,0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,-3,3,0,0,-2,-1,0,0,
        9,-9,9,-9,6,3,-3,-6,6,-6,-3,3,4,2,1,2,
        -6,6,-6,6,-4,-2,2,4,-3,3,3,-3,-2,-1,-1,-2,
        2,-2,0,0,1,1,0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,2,-2,0,0,1,1,0,0,
        -6,6,-6,6,-3,-3,3,3,-4,4,2,-2,-2,-2,-1,-1,
        4,-4,4,-4,2,2,-2,-2,2,-2,-2,2,1,1,1,1};
    int l,k,j,i;
    float xx,d1d2,cl[16],x[16];

    d1d2=d1*d2;
    for (i=1;i<=4;i++) {          压缩暂时向量 x
        x[i-1]=y[i];
        x[i+3]=y1[i]*d1;
        x[i+7]=y2[i]*d2;
        x[i+11]=y12[i]*d1d2;
    }
    for (i=0;i<=15;i++) {         矩阵乘以存好的表
        xx=0.0;
        for (k=0;k<=15;k++) xx += wt[i][k]*x[k];
        cl[i]=xx;
    }
    l=0;
    for (i=1;i<=4;i++)            将结果解压缩成输出表
        for (j=1;j<=4;j++) c[i][j]=cl[l++];
}
```

等式(3.6.6)求双三次插值，其实现非常简单。它调用上面的程序**bcucof**，并返回插值函数值及两个梯度值：

```
#include "nrutil.h"

void bcuint (float y[], float y1[], float y2[], float y12[], float x1l, flaot x1u, float x2l, float x2u, float
        x1, float x2, float *ansy, float *ansy1, float *ansy2)
```

在方格坐标区域内求双三次插值。输入量包括y,y1,y2,y12(跟**bcucof**要求的一样)；x1l和x1u为在第一个方向上，方形坐标区域的上下界坐标；x2l和x2u同理为第二个方向上的上下界坐标；x1,x2为所求插值点的坐标；ansy为返回的插值函数值，ansy1和ansy2为插值梯度值。本程序调用**bcucof**。

```
{
    void bcucof(float y[], float y1[], float y2[], float y12[], float d1,
        float d2, float **c);
    int i;
    float t,u,d1,d2,**c;
```

```
    c=matrix(1,4,1,4);
    d1=x1u-x1l;
    d2=x2u-x2l;
    bcucof(y,y1,y2,y12,d1,d2,c);        得到c值
    if (x1u == x1l || x2u == x2l) nrerror("Bad input in routine bcuint");
    t=(x1-x1l)/d1;                      等式(3.6.4)
    u=(x2-x2l)/d2;
    *ansy=(*ansy2)=(*ansy1)=0.0;
    for (i=4;i>=1;i--) {                等式(3.6.6)
        *ansy=t*(*ansy)+((c[i][4]*u+c[i][3])*u+c[i][2])*u+c[i][1];
        *ansy2=t*(*ansy2)+(3.0*c[i][4]*u+2.0*c[i][3])*u+c[i][2];
        *ansy1=u*(*ansy1)+(3.0*c[4][i]*t+2.0*c[3][i])*t+c[2][i];
    }
    *ansy1 /= d1;
    *ansy2 /= d2;
    free_matrix(c,1,4,1,4);
}
```

### 3.6.3 用高维插值获得高平滑度:双三次样条

要获得二维插值高平滑度的另一种常用技术是**双三次样条**。实际它等于是双三次插值的一种特殊情况:插值函数与等式(3.6.6)的函数形式完全相同,不过坐标方格点的导数值是由一维样条"全局"确定的。然而,双三次样条通常的实现形式,看起来跟上面的双三次插值程序相差甚远,倒是更接近上面的 **polin2**程序的形式:求函数插值时,先按列表的行方向求 m 次一维样条插值,再按新建的列方向做一次一维样条插值。这样做在所需的预计算和存储等方面都很得体(时间和空间的权衡)。没有(象双三次插值那样)需预先计算并存好所有的导数信息,而是象样条插值一样,只需先计算并保存一张辅助表,它是仅沿一个方向上的二阶导数。接下来,只需对 m 行样条做样条求值(不用构造),还要对最后的列样条做一次构造并且求值。(回忆一下,构造样条的复杂度为 $N$ 阶,而样条求值仅为 $\log N$ 就可查到表中的位置!)

这里的程序是预先计算辅助的二阶导数表:

```
void splie2(float x1a[], float x2a[], float **ya, int m, int n, float **y2a)
    给定列表函数ya[1..m][1..n]及列表独立变量x1a[1..m]和x2a[1..n],本程序构造ya的行的一维自然三次样
    条,返回的二阶导数存于数组y2a[1..m][1..n]中。
{
    void spline(float x[], float y[], int n, float yp1, float ypn, float y2[]);
    int j;

    for (j=1;j<=m;j++)
        spline(x2a,ya[j],n,1.0e30,1.0e30,y2a[j]);        值1×10^30标志自然样条
}
```

(如果想要对更大矩阵的子块插值,参考第1.2节)。

上面的程序执行过一次以后,可以连续调用下面的程序执行任意次求双三次样条插值:

```
#include "nrutil.h"

void splin2(float x1a[], float x2a[], float **ya, float **y2a, int m, int n, float x1, float x2,
        float *y)
    给定x1a,x2a,ya,m,n,它们的说明见程序splie2,并由该程序生成y2a,对给定所求的插值点x1,x2,本程序由双三
    次样条插值求得插值函数值y

    void spline(float x[], float y[], int n, float yp1, float ypn, float y2[]);
    void splint(float xa[], float ya[], float y2a[], int n, float x, float *y);
```

```
int j;
float *ytmp,*yytmp;

ytmp=vector(1,m);
yytmp=vector(1,m);                                  使用一维样条求值程序 splint，
for (j=1;j<=m;j++)                                  采用 splie2 构造的行样条执行 m 个求值计算
    splint(x2a,ya[j],y2a[j],n,x2,&yytmp[j]);
spline(x1a,yytmp,m,1.0e30,1.0e30,ytmp);             构造一维列样条并求值计算
splint(x1a,yytmp,ytmp,m,x1,y);
free_vector(yytmp,1,m);
free_vector(ytmp,1,m);
}
```

# 第四章 函数积分

## 4.0 引言

数值积分方法又称**求积法**(quadrature)，其历史要追溯到微积分发明之前，或甚至更早的时候。初等函数的导数一般能解析计算出来，而积分则不能，这使得数值积分这一学科的作用更为显著，而不必象在整个十八、十九世纪中那样，作为数值分析的算法研究。

随着自动计算的发明，求积分仅成为一项并不很有趣的数值计算工作了。自动计算，即使是最原始的形式，如使用台式计算器和能装满一间房子的计算机(直到五十年代中还是人而不是机器)在内，使微分方程的数值积分这一极为丰富的领域展示了可能性。求积分仅是其中一种最简单的特例：积分值

$$I = \int_a^b f(x)dx \tag{4.0.1}$$

与对微分方程(4.0.2)求 $I \equiv y(b)$ 的值等价：

$$\frac{dy}{dx} = f(x) \tag{4.0.2}$$

上式有边界条件

$$y(a) = 0 \tag{4.0.3}$$

本书第十六章将讨论微分方程的数值积分，届时着重强调步长的"可变"选择，或"自适应"选择的概念。这里不想进一步探讨，如果待积分的函数明显地集中在一个或几个峰值点处，或其形状很难用单个长度标尺描述，那么应该将问题化成(4.0.2)和(4.0.3)的形式，然后再采用第十六章方法。

本章求积分法的原理就是，以一定方式在积分区间内，按横坐标顺序将被积函数的函数值加起来。目标是用最少的求被积函数值的次数，而得到尽可能精确的积分值。和内插法(第三章)的情况类似，可自由选择不同阶的各种方法，有时(但不总是)越高阶的精度越高。第4.3节讨论的"龙贝格(Romberg)积分"，是利用各种不同阶的积分方法的一般形式，它是一种值得推崇的方法。

除了本章和十六章的方法外，求积分的方法很多。其中很重要的一种是利用逼近函数。第5.9节将仔细讨论切比雪夫(Chebyshev)逼近函数积分。读者应了解如何利用第3.3节中 **spline** 程序的结果，来求**三次样条函数**的积分，这里将不作详细讨论。(提示：用式(3.3.3)对 $x$ 解析积分。参见[1])

傅里叶变换中涉及的积分可由快速傅里叶变换(FFT)算法算出。这在第13.9节中讨论。多重积分内容也很多，第4.6节将作一般性讨论。比较重要的蒙特—卡罗(Monte－Carlo)积分方法在第七章里介绍。

参考文献和进一步读物：

Kahaner, D., Moler, C., and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs, NJ: Prentice Hall), Chapter 5.

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §5.2, p. 89. [1]

Davis, P., and Rabinowitz, P. 1984, *Methods of Numerical Integration*, 2nd ed. (Orlando, FL: Academic Press).

## 4.1 坐标等距划分的经典公式

哪本关于数值分析的书上，会没有谈及辛卜生(Simpson)先生和他的"法则"呢？当一个函数在等距点处的值已知，求这个函数积分的经典公式是极精巧雅致的，它让人们联想到它的历史。正是通过这些公式，现代数值分析家们可以返回到数个世纪前甚至远至牛顿时代的人们，与他或她们的前辈交流思想。但时代的确在变化，除了其中最根本的两个公式("扩展梯形法"式(4.1.1)和"扩展中点法"式(4.1.19)，见第4.2节)以外，这些经典公式几乎都没什么用处了，可以进博物馆了，但它们确是很精美的展品。

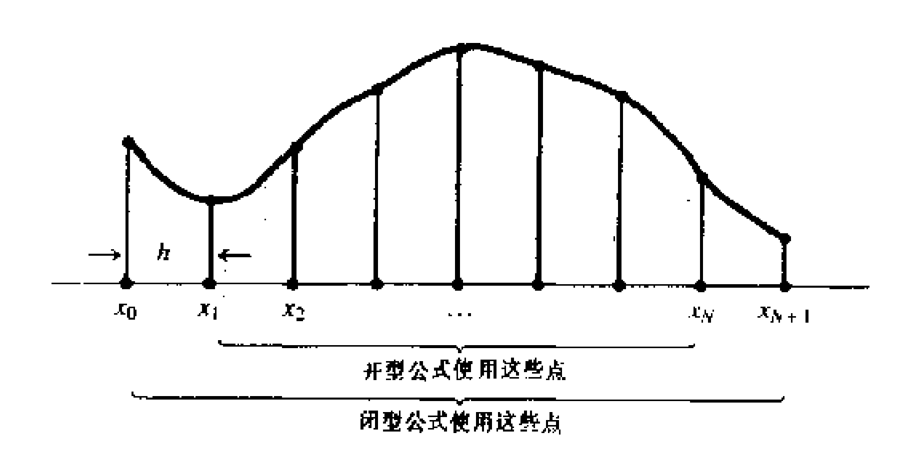
先作一些约定：一组坐标序列 $x_0,x_1,\cdots,x_N,x_{N+1}$，其间隔为定步长 $h$，

$$x_i = x_0 + ih \qquad i = 0,1,\cdots,N+1 \tag{4.1.1}$$

已知函数 $f(x)$ 在各 $x_i$ 处值

$$f(x_i) = f_i \tag{4.1.2}$$

求函数 $f(x)$ 在下限 $a$ 和上限 $b$ 之间的积分，其中 $a$，$b$ 分别等于其中某一个 $x_i$ 值。使用了在端点的函数值 $f(a)$、$f(b)$ 的积分公式，称为闭型公式。有时被求积分的函数在一个或两个端点处的函数值难以计算(如求 $f$ 变成求%型极限，或者更糟糕的是，端点处为一可积奇异性。)此时，就需要一个开型公式，即只用严格限于 $a$、$b$ 之间的 $x_i$ 值来估计积分(见图4.1.1)。




闭型公式要计算在边界点处的值，而开型公式不这样做(以防边界点上的求值使算法失效)。

图4.1.1 用等距划分的坐标来求函数从 $x_0$ 到 $x_{N+1}$ 之间积分的求积公式。

经典公式的基本构造模块是，计算函数在一个小区间上的积分的"方法"，区间数目增多时，就会发现这些方法对阶数逐渐增高的多项式是精确的(记住在实数情况下，高阶并非意

谓着高精度）。下面给出一些这样的闭型公式。

### 4.1.1 牛顿-柯特斯（Newton-Cotes）闭型公式

梯形法：

$$\int_{x_1}^{x_2} f(x)dx = h\left[\frac{1}{2}f_1 + \frac{1}{2}f_2\right] + O(h^3 f'') \qquad (4.1.3)$$

其中误差项$O()$表示真实值与估计值之差，其大小等于某一数值系数乘以$h^3$，再乘以函数在被积区间内某一点处的二阶导数的积。这个数值系数是已知的，可在任何一本关于这一内容的标准参考资料中找到。但应在哪一点求二阶导数却是未知的。要是我们能知道这一点，就能求出该点的值。这样的话，这种方法还真不错！由于一个已知数与一个未知数之积仍是一个未知数，因此我们还照原样书写公式，只写误差项$O()$，而不写出数值系数。

式(4.1.3)为一个两点公式($x_1$及$x_2$)。对小于等于一次的多项式是精确的，如$f(x)=x$。有人希望能有一个适用于小于等于二次多项式的三点公式。确实有这样的公式，而且由于公式左右对称引起系数相消，这种三点公式对小于等于三次的多项式仍是精确的，如$f(x)=x^3$。

辛卜生(Simpson)法：

$$\int_{x_1}^{x_3} f(x)dx = h\left[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{1}{3}f_3\right] + O(h^5 f^{(4)}) \qquad (4.1.4)$$

其中$f^{(4)}$表示函数在积分区间某未知点处的四阶导数值。还要注意，公式给出的是距离为$2h$的区间上的积分，所以系数加起来等于2。

四点公式里的系数相消就不怎么好了，但对小于等于三次的多项式也是精确的。

辛卜生3/8法：

$$\int_{x_1}^{x_4} f(x)dx = h\left[\frac{3}{8}f_1 + \frac{9}{8}f_2 + \frac{9}{8}f_3 + \frac{3}{8}f_4\right] + O(h^5 f^{(4)}) \qquad (4.1.5)$$

五点公式就又有消去性质了。

波特(Bode)法

$$\int_{x_1}^{x_5} f(x)dx = h\left[\frac{14}{45}f_1 + \frac{64}{45}f_2 + \frac{24}{45}f_3 + \frac{64}{45}f_4 + \frac{14}{45}f_5\right] + O(h^7 f^{(6)}) \qquad (4.1.6)$$

上式对小于等于五次的多项式是精确的。

至此，公式不再以人名命名，这里也不再接着探讨下去，后面按顺序其它公式可参阅文献[1]。

### 4.1.2 单个区间的外推公式

我们暂时偏离历史发展的顺序。在这里，大多数书上会给出一些"Newton-Cotes 开型公式"，例如：

$$\int_{x_0}^{x_5} f(x)dx = h\left[\frac{55}{24}f_1 + \frac{5}{24}f_2 + \frac{5}{24}f_3 + \frac{55}{24}f_4\right] + O(h^5 f^{(4)})$$

注意，上式是求从$a=x_0$到$b=x_5$，却只用了内部各点$x_1,x_2,x_3,x_4$。这种开型公式用处不大

原因是：(i)不能串起来构成有用的扩展方法，就象我们将处理闭型公式那样；(ii)其它用处都受到高斯积分公式的支配，将在第4.5节中介绍高斯求积公式。

现在，我们不从牛顿-柯特斯公式着手，而从$x_0$到$x_1$的单区间内估计积分的公式着手，这一积分用$f$在$x_1$、$x_2$、…处的值表示。

$$\int_{x_0}^{x_1} f(x)dx = h[f_1] + O(h^2 f') \tag{4.1.7}$$

$$\int_{x_0}^{x_1} f(x)dx = h\left[\frac{3}{2}f_1 - \frac{1}{2}f_2\right] + O(h^3 f'') \tag{4.1.8}$$

$$\int_{x_0}^{x_1} f(x)dx = h\left[\frac{23}{12}f_1 - \frac{16}{12}f_2 + \frac{5}{12}f_3\right] + O(h^4 f^{(3)}) \tag{4.1.9}$$

$$\int_{x_0}^{x_1} f(x)dx = h\left[\frac{55}{24}f_1 - \frac{59}{24}f_2 + \frac{37}{24}f_3 - \frac{9}{24}f_4\right] + O(h^5 f^{(4)}) \tag{4.1.10}$$

有人也许会问上面公式怎样得到的，有很多精美的方法，但最直接的方法是，在公式的基本形式中，用未知数，比如$p$、$q$、$r$、$s$代替其中的系数。不失一般性，取$x_0=0$、$x_1=1$，则$h=1$。依次用$f(x)=1$、$f(x)=x$、$f(x)=x^2$、$f(x)=x^3$代换$f(x)$（还有$f_1$、$f_2$、$f_3$、$f_4$），每次求一次积分，使右边成为一个数，而左边成为未知数$p$、$q$、$r$、$s$的线性方程。求解由此得到的四个方程，即可求得系数。

### 4.1.3 扩展公式（闭型）

使用方程(4.1.3)$N-1$次，计算区间$(x_1,x_2)$，$(x_2,x_3)$…，$(x_{N-1},x_N)$中的积分，再把结果加起来，得到从$x_0$到$x_N$积分的“扩展”或“合成”公式。

**扩展梯形法**：

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{1}{2}f_1 + f_2 + f_3 + \cdots + f_{N-1} + \frac{1}{2}f_N\right] + O\left(\frac{(b-a)^3 f''}{N^2}\right) \tag{4.1.11}$$

这里误差项用了区间长度$b-a$和点数$N$来表示，而不用$h$。通常$a$、$b$固定，若取步长个数为原来的两倍，那么误差项将减少多少，这一点从公式看就很清楚了（此时按4倍关系改变）。所以在后面公式中，误差项只表示为步长个数的比例因子。

事实上，方程(4.1.11)是本节最重要的公式，是大多数实用积分方法的的基础，这一点到第4.2节才会明白。

$1/N^3$**阶的扩展公式**为：

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{5}{12}f_1 + \frac{13}{12}f_2 + f_3 + f_4 + \cdots + f_{N-2} + \frac{13}{12}f_{N-1} + \frac{5}{12}f_N\right] + O\left(\frac{1}{N^3}\right) \tag{4.1.12}$$

（等一会说明怎么得到的）。

对连续的、不相重叠的区间对，应用式(4.1.22)得到**扩展辛卜生法**：

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{2}{3}f_3 + \frac{4}{3}f_4 + \right.$$

$$\cdots - \frac{2}{3}f_{N-2} + \frac{4}{3}f_{N-1} + \frac{1}{3}f_N\Big] + O\left(\frac{1}{N^4}\right) \qquad (4.1.13)$$

注意，上式中括号内2/3、4/3一直交替出现。有人会以为这种摆动的交替过程，是否包含[illegible]于被积函数积分的某种不易察觉的性质。事实上，这种交替只是由于用了式(4.1.4)构造模式的结果。

与辛卜生法同阶的还有另外一个扩展公式：

$$\int_{x_1}^{x_N} f(x)dx = h\Big[\frac{3}{8}f_1 + \frac{7}{6}f_2 + \frac{23}{24}f_3 + f_4 + f_5 + \cdots + f_{N-4} + f_{N-3} + \frac{23}{24}f_{N-2} + \frac{7}{6}f_{N-1} + \frac{3}{8}f_N\Big] + O\left(\frac{1}{N^4}\right) \qquad (4.1.14)$$

本方程是通过对每连续四个一组的点集上拟合三次多项式得到。关于这一点的细节延至第18.3节中讨论，该节中为解积分方程使用了相似的方法。现在，我可以说明式(4.1.12)怎么得到的了。在第一个区间及最后一个区间里，使用式(4.1.3)的梯形法，其它区间用辛卜生法，这样得到的改进结果再与直接使用辛卜生扩展法得到的结果取平均。梯形法比辛卜生法低两阶，这样使得积分的阶次降为另外一个 $N$ 的幂(因为梯形法只用了两次而不是 $N$ 次)。因此，这使得到的公式比辛卜生法低一阶。

### 4.1.4 扩展公式(开型与半开型)

式(4.1.11)～式(4.1.14)加上式(4.1.7)～式(4.1.10)，即可以推导出开型的或半开型的公式，其中将式(4.1.7)～式(4.1.10)用于第一步的外推的开型公式，而式(4.1.11)～式(4.1.14)用于求第二步及后面各步值的闭型公式。正如上面刚刚讨论过的那样，在区间边界求积分的各步，可选用比区间之内各步低一阶公式。这样，所得区间两端都为开型的公式如下：

由式(4.1.7)和式(4.1.11)得到

$$\int_{x_1}^{x_N} f(x)dx = h\Big[\frac{3}{2}f_2 + f_3 + f_4 + \cdots + f_{N-2} + \frac{3}{2}f_{N-1}\Big] + O\left(\frac{1}{N^2}\right) \qquad (4.1.15)$$

由式(4.1.8)及(4.1.12)得到

$$\int_{x_1}^{x_N} f(x)dx = h\Big[\frac{23}{12}f_2 + \frac{7}{12}f_3 + f_4 + f_5 + \cdots + f_{N-3} + \frac{7}{12}f_{N-2} + \frac{23}{12}f_{N-1}\Big] + O\left(\frac{1}{N^3}\right) \qquad (4.1.16)$$

由式(4.1.9)及(4.1.13)得到

$$\int_{x_1}^{x_N} f(x)dx = h\Big[\frac{27}{12}f_2 + 0 + \frac{13}{12}f_4 + \frac{4}{3}f_5 + \cdots + \frac{4}{3}f_{N-4} + \frac{13}{12}f_{N-3} + 0 + \frac{27}{12}f_{N-1}\Big]$$

$$+O\left(\frac{1}{N^4}\right) \tag{4.1.17}$$

区间内部各点系数由4/3,2/3交替变化。为消去这种交替变化,由式(4.1.9)及式(4.1.14)可得

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{55}{24}f_2 - \frac{1}{6}f_3 + \frac{11}{8}f_4 + f_5 + f_6 + f_7 + \cdots + f_{N-5} + f_{N-4} + \frac{11}{8}f_{N-3} - \frac{1}{6}f_{N-2} + \frac{55}{24}f_{N-1}\right] + O\left(\frac{1}{N^4}\right) \tag{4.1.18}$$

再简单提一下另一种扩展开型公式,它用于积分限位于标定坐标中点的的情况,一般称为**扩展中点法**,其精度如式(4.1.15)是二阶的

$$\int_{x_1}^{x_N} f(x)dx = h[f_{3/2} + f_{5/2} + f_{7/2} + \cdots + f_{N-3/2} + f_{N-1/2}] + O\left(\frac{1}{N^2}\right) \tag{4.1.19}$$

这种情况也有高阶公式,这里就不作介绍了。

**半开型公式**就是式(4.1.11)~式(4.1.14)分别与式(4.1.15)~式(4.1.18)的线性组合。积分的闭端使用前四个式子中的权;而积分的开端使用后四个式子中的权。举例说明,一个右闭左开误差按$1/N^3$减小的公式为:

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{23}{12}f_2 + \frac{7}{12}f_3 + f_4 + f_5 + \cdots + f_{N-2} + \frac{13}{12}f_{N-1} + \frac{5}{12}f_N\right] + O\left(\frac{1}{N^3}\right) \tag{4.1.20}$$

参考文献和进一步读物:


Abramowitz, M., and Stegun, I. A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, vol. 55, § 25.4. [1]

Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods* (New York: Wiley), § 7.1.


## 4.2 基本算法

我们从式(4.1.11)的梯形法着手。至于为什么要从梯形法开始来研究众多其它算法,原因有两个,一个很显然,另一个深奥些。

显然的原因是,在固定积分限 $a,b$ 之间求固定函数的积分时,可以将梯形法中分划区间的数目增加一倍后,仍然可以利用前面工作的结果。实现梯形法最粗糙的步骤是,在积分区域端点 $a,b$ 对函数取平均。而精细时的第一步是,将中点处函数值加入到那个平均值,第二步加进1/4和3/4点处的函数值。如此等等(见图4.2.1)。

不费多大力气可写出程序实现这种算法:

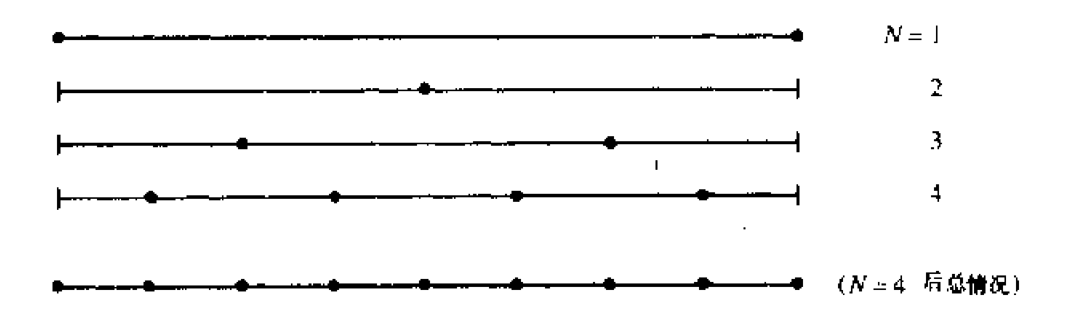
```
#define FUNC(x) ((*func)(x))

float trapzd(float (*func)(float), float a, float b, int n)
    本程序计算扩展梯形法第n次的加工结果。func为输入,是指向被积函数的指针,函数积分限a,b也为输入,n=1时
```

调用，返回 $\int_a^b f(x)dx$ 的粗略估计。随后 $n=2,3,\cdots$ 调用时，每次增加 $2^{n-2}$ 个内点以改善精度。

```
{
    float x, tnm, sum, del;
    static float s;
    int it, j;

    if (n == 1) {
        return (s=0.5 * (b-a) * (FUNC(a)+FUNC(b)));
    } else {
        for (it=1,j=1;j<n-1;j++) it <<= 1;
        tnm=it;
        del=(b-a)/tnm;                                   这是所增加点的间隔
        x=a+0.5*del;
        for (sum=0.0,j=1;j<=it;j++,x+=del) sum += FUNC(x);
        s=0.5*(s+(b-a)*sum/tnm);                        用修正值替代S
        return s;
    }
}
```




顺序调用程序 **trapzd**，每次可利用前面各次的结果，精细时仅须在新增点处求被积函数值。最下面一条线表示经过4次调用后求函数值的点数。程序 **qsimp** 为中间结果加权，这样无须多加修改就可由梯形法转变为辛卜生法。

图4.2.1

调用上面程序 **trapzd** 有很多方式。最简单的一种就是，事先知道应分划多少区间（多少步）的情况下（不能想像！）求一个函数的积分。若须 $2^m+1$ 步，可用下面程序完成，返回值 s

```
for(j=1;j<=m+1;j++) s=trapzd(func,a,b,j)
```

当然更好的办法是改进梯形法，以达到某种特殊的精度要求。

```
#include  <math.h>
#define  EPS 1.0e-5
#define  JMAX   20

float qtrap(float (*func)(float), float a, float b)
    从a到b返回函数func的积分值。参数EPS可设置为要求的相对精度，同时设置JMAX，使允许的最多步数为2^(JMAX-1)
    -1，用梯形法求积分。
{
    float trapzd(float (*func)(float), float a, float b, int n);
    void nrerror(char error_text[]);
    int j;
    float s,olds;
```

```
    olds = -1.0e30;                                这里任何数都行，只要不是函数在端点处的平均值
    for (j=1;j<=JMAX;j++) {
        s=trapzd(func,a,b,j);
        if (fabs(s-olds) < EPS*fabs(olds)) return s;
        olds=s;
    }
    nrerror("Too many steps in routine qtrap");
    return 0.0;                                    永远不到此
}
```

**qtrap** 虽然不很复杂，但用于求一般不很平滑函数的积分很有力。复杂性增加后，往往可变换成高阶方法，但高阶方法仅当被积函数足够平滑时才有效果。若被积函数的变量是测量数据点之间的线性插值时，则可选用 **qtrap** 程序方法。注意对 EPS 不能要求过于严格，因为若 **qtrap** 为达到要求的精度而迭代步数过多时，累积的舍入误差会增大，程序就不能收敛了。一般 32 位机 $10^{-6}$的精度可能刚好临界，当收敛速度适中时该值可以达到，否则达不到。

现在谈谈，那个关于式(4.1.11)"深奥"一些的原因。这个原因就是，表达式中误差项以$1/N^2$阶开始，表示成$1/N$ 的阶次形式总是偶数阶的，由 Euler-Maclaurin 公式可直接看出：

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{1}{2}f_1 + f_2 + f_3 + \cdots + f_{N-1} + \frac{1}{2}f_N\right] - \frac{B_2h^2}{2!}(f'_N - f'_1) - \cdots - \frac{B_{2k}h^{2k}}{(2k)!}(f_N^{(2k-1)} - f_1^{(2k-1)}) - \cdots \tag{4.2.1}$$

其中 $B_{2k}$为**伯努利数**(Bernoulli number)，由下面生成函数给出。

$$\frac{t}{e^t - 1} = \sum_{n=0}^{\infty} B_n \frac{t^n}{n!} \tag{4.2.2}$$

前几个偶数项值(除 $B_1 = -1/2$外，奇数项都为零)为：

$$B_0 = 1 \qquad B_2 = \frac{1}{6} \qquad B_4 = -\frac{1}{30} \qquad B_6 = \frac{1}{42}$$
$$B_8 = -\frac{1}{30} \qquad B_{10} = \frac{5}{66} \qquad B_{12} = -\frac{691}{2730} \tag{4.2.3}$$

式(4.2.1)并非是一个收敛展开式，仅为一个近似展开式，表示式中不论从哪一项开始截去，其后面项产生的误差总是小于略去项中第一项的两倍。不收敛的原因是伯努利数会变得很大，如

$$B_{50} = \frac{495057205241079648212477525}{66}$$

关键是，式(4.2.1)中误差级数只有 $h$ 的偶数次幂。一般说来，第4.1节的高阶求积分方法并不具备这种性质。例如，(4.1.13)式误差的级数就是从 $O(1/N^3)$开始，但紧接着还有其它 $N$ 的幂次，如$1/N^4$，$1/N^5$等等。

设用 $N$ 个分划($N$ 步)求式(4.1.11)的值，结果为 $S_N$。再由$2N$ 步得到 $S_{2N}$(只须接连调用 **trapzd** 两次)。第二次结果的首项误差只是第一次的$1/4$，故组合式

$$S = \frac{4}{3}S_{2N} - \frac{1}{3}S_N \tag{4.2.4}$$

就会消去误差的首项，故式(4.2.1)中$1/N^2$阶的误差项就不存在。剩下的误差就是$1/N^4$阶的，与辛卜生法相同。事实上，不久就会看到式(4.2.4)其实正是辛卜生法，系数2/3，4/3交替变化。用辛卜生法求积分，这种方法更可取。由此得到一个与上面 **qtrap** 极其类似的程序

```
#include <math.h>
#define EPS 1.0e-6
#define JMAX 20

float qsimp (float (*func)(float), float a, float b)
    返回函数func从a到b的积分。参数EPS设置为要求的相对精度，同时设置JMAX，使得允许的最大步数为2^(JMAX-1)，用辛卜生法求积分。
{
    float trapzd(float (*func)(float), float a, float b, int n);
    void nrerror(char error_text[]);
    int j;
    float s,st,ost,os;

    ost = os = -1.0e30;
    for (j=1;j<=JMAX;j++) {
        st=trapzd(func,a,b,j);
        s=(4.0*st-ost)/3.0;                          与上面式(4.2.4)比较
        if (fabs(s-os) < EPS*fabs(os)) return s;
        os=s;
        ost=st;
    }
    nrerror("Too many steps in routine qsimp");
    return 0.0;                                      永远不到此
}
```

一般若被积函数的四阶导数有界(即三阶导数连续)，则 **qsimp** 比 **qtrap** 效果好(即要求的函数值次数较少)。有时，对于一些轻易的工作，此时将 **qsimp** 与它必需部分 **trapzd** 结合起来是一种好方法。

## 4.3 龙贝格积分

可以将龙贝格(Romberg)积分看成是上节程序 **qsimp** 的自然推广，其积分格式比辛卜生法具有更高的阶次。其基本思想是，利用扩展梯形法中连续进行 $k$ 次细分的结果(由 **qtrap** 实现)，以消除误差级数直到(不包括)$O(N^{2k})$的所有项。**qsimp** 是 $k=2$的特例。这是一例关于通常称为**理查德森**(Richardson)**延迟极限逼近**的一般思想：即对不同的参数值 $h$ 完成数值算法，然后将结果外推到连续极限 $h=0$。

式(4.2.4)中减去了误差首项，是多项式外推法的一个特例。在更一般的龙贝格情形下，我们还可以采用尼维尔(Neville)算法(第3.1节)，将其逐次地精确外推到零步长的情形。在龙贝格积分程序中，能很简洁地编写尼维尔算法。为使程序简洁起见，调用第3.1节给出的函数 **polint** 来做外推，似乎要更好些。

```
#include <math.h>
#define EPS 1.0e-6                 这里EPS为要求的精度，由外推误差估算确定
#define JMAX 20                    JMAX限制步长总数
#define JMAXP (JMAX+1)
#define K 5                        K是用于外推的点数
```

```
float qromb(float (*func)(float),float a,float b)
    返回函数 func 从 a 到 b 的积分值，积分采用龙贝格法，阶数为2k，当 k=2为辛卜生法。
{
    void polint(float xa[], float ya[], int n, float x, float *y, float *dy);
    float trapzd(float (*func)(float), float a, float b, int n);
    void nrerror(char error_text[]);
    float ss,dss;
    float s[JMAXP+1],h[JMAXP+1];              存储连续梯形法逼近及其相应步长
    int j;

    h[1]=1.0;
    for (j=1;j<=JMAX;j++) {
        s[j]=trapzd(func,a,b,j);
        if (j >= K) {
            polint(&h[j-K],&s[j-K],K,0.0,&ss,&dss);
            if (fabs(dss) < EPS * fabs(ss)) return ss;
        }
        s[j+1]=s[j];
        h[j+1]=0.25*h[j];      关键步，步长减为0.5倍而因子却是0.25，这样使得按(4.2.1)式外推到h²的多
                               项式，而不仅仅是h的多项式
    }
    nrerror("Too many steps in routine qromb");
    return 0.0;                永远不到此
}
```

**qromb** 程序，连同必需的 **trapzd** 及 **polint**，用于求足够平滑（解析）函数，在不含有奇点且端点为非奇点的区域上的积分，是很有效的。此时 **qromb** 求函数值的次数远少于第4.2节中的任一种方法。例如，积分

$$\int_0^2 x^4\log(x+\sqrt{x^2+1})dx$$

**qromb** 程序经五次调用 **trapzd** 后，第一次外推就收敛，而同样的积分，**qsimp** 需调用8次 **trapzd**，而 **qtrap** 需调用13次。

## 4.4 广义积分

就目前的目的而言，称满足下列条件之一的积分为"广义"的：

- 在积分上下限处函数有有限极限值，但正好在端点之一处不能直接求函数值（如 $\sin(x)/x$ 在 $x=0$处）。
- 上限为 $+\infty$，或下限为 $-\infty$。
- 区间两端有可积奇异点（如 $x^{-1/2}$在 $x=0$处）。
- 上下限之间某一已知点为可积奇异点。
- 上下限之间某一未知点为可积奇异点。

若积分是无穷大（如 $\int_1^{\infty} x^{-1}dx$），或在极限意义下不存在（如 $\int_{-\infty}^{\infty}\cos x dx$），都不能称为广义的，而称为不可积。若本身是不可积的积分，算法再多也没有用。

本节将对上两节的方法进行推广，使其对上面列出的前四个情形仍有效，至于关于不可知奇异点的第五个问题，则只能用可变步长的微分方程积分算法，这个问题留给第十六章。

首先，必须有一个象扩展梯形法式(4.1.11)那样的基本方法，但必须按第4.1节的约定是开型公式，即不必在端点处求被积函数的值。最好选用扩展中点法的方程式(4.1.19)，原

因是式(4.1.19)与式(4.1.11)一样，具有误差级数都是 $h$ 的偶次幂的性质。有一个不太有名的公式，称为 Euler - Maclaurin **第二求和公式**：

$$\int_{x_1}^{x_N} f(x)dx = h[f_{3/2} - f_{5/2} + f_{7/2} - \cdots + f_{N-3/2} + f_{N-1/2}] + \frac{B_2 h^2}{4}(f'_N - f'_1) + \cdots + \frac{B_{2k} h^{2k}}{(2k)!}(1 - 2^{-2k+1})(f_N^{(2k-1)} - f_1^{(2k-1)}) + \cdots \tag{4.1.1}$$

这个公式可由式(4.2.1)导出，先用步长 $h$ 代入式(4.2.1)，再用 $h/2$代入，然后用第一次结果减去第二次结果的两倍即可。

扩展中点法不可能将步数增加一倍而仍可以利用前面函数求值的结果(试试看?)。但将步数增加到原来的三倍却是可以的。是增加到三倍好，还是增加到两倍而承认这种损失好呢?一般说来，增加到三倍后不必要的工作也增加到 $\sqrt{3}$ 倍，因为达到期望的精度所必需的步数，可能落在三倍引起的对数区间的任何地方。而增加到二倍的话，这个因子仅为 $\sqrt{2}$，但因不能利用前面计算结果故而丢掉了一个额外因子2。由于1.732<2×1.414，得知步长数增加到原来的三倍要好些。

下面就是要求的程序，可与 **trapzd** 相媲美

```
#define FUNC(x) ((*func)(x))

float midpnt(float (*func)(float), float a, float b, int n)
    本程序计算扩展中点法第 n 阶段的精细结果。func 为输入，是指向待积分函数的指针。积分限 a,b 也为输入。当 n=1
    调用时返回 ∫_a^b f(x)dx 的粗略估计，随后 n=2,3…调用(按顺序)时，每次增加(2/3)×3^(n-1)个内点以改善精度
{
    float x,tnm,sum,del,ddel;
    static float s;
    int it,j;

    if (n == 1) {
        return (s=(b-a)*FUNC(0.5*(a+b)));
    } else {
        for(it=1,j=1;j<n-1;j++) it *= 3;
        tnm=it;
        del=(b-a)/(3.0*tnm);
        ddel=del+del;                       增加的点交替地在del和ddel之间划分
        x=a+0.5*del;
        sum=0.0;
        for (j=1;j<=it;j++) {
            sum += FUNC(x);
            x += ddel;
            sum += FUNC(x);
            x += del;
        }
        s=(s+(b-a)*sum/tnm)/3.0;            由新旧积分的组合得到精细的积分值
        return s;
    }
}
```

在第4.2节的 **qtrap** 驱动程序中，可用 **midpnt** 替换 **trapzd**，可很容易将 **trapzd**(func,a,b,j)改为 **midpnt**(func,a,b,j)，可能也要减小参数 JMAX，因为$3^{JMAX-1}$(三倍)比$2^{JMAX-1}$(二倍)

大得多。

实现开型公式类似于辛卜生法(第4.2节中的 **qsimp**)，只须用 **midpnt** 替换 **trapzd**，再如上述减小 JMAX，现将外推步也改为

s=(9.0*st—ost)/8.0

因为三倍步数后，误差小到1/9大小，而不象两倍时那样减小到1/4。

无论是修改后的 **qtrap** 还是 **qsimp**，都适用于本节开头列出的第一类问题。更复杂的是按下面方式对龙贝格积分进行推广：

```
#include <math.h>
#define EPS 1.0e-6
#define JMAX 14
#define JMAXP (JMAX+1)
#define K 5

float qromo(float (*func)(float), float a, float b,
       float(*choose)(float(*)(float), float, float, int))
```

开区间上的龙贝格积分，采用任意指定的积分子程序 **choose** 及龙贝格方法，返回函数 func 从 a 到 b 的积分。**choose** 应为开型公式，不用求函数在端点处的值，**choose** 每次调用时将步数增加三倍，其误差级数只含有步数的偶次幂项。**choose** 可以选用 **midpnt midinf midsql midsqu** 中任一程序

```
{
    void polint(float xa[], float ya[], int n, float x, float *y, float *dy);
    void nrerror(char error_text[]);
    int j;
    float ss,dss,h[JMAXP+1],s[JMAXP+1];

    h[1]=1.0;
    for (j=1;j<=JMAX;j++) {
        s[j]=(*choose)(func,a,b,j);
        if (j >= K) {
            polint(&h[j-K],&s[j-K],K,0.0,&ss,&dss);
            if (fabs(dss) < EPS*fabs(ss)) return ss;
        }
        s[j+1]=s[j];
        h[j+1]=h[j]/9.0;          这里步长增加三倍，并误差级数为偶次
    }
    nrerror("Too many steps in routing qromo");
    return 0.0;                 永远不达到这里
}
```

不要对 **qromo** 程序中难懂的参数说明而烦恼。下例是一种典型的调用形式(见塞尔函数从0至2的积分)：

```
#include "nr.h"
float answer ;
...
anwer=qromo(bessy0,0.0,2.0,midpnt);
```

**qromo** 与 **qromb** 几乎完全一样，似乎没有必要把 **qromo** 全部写出来。但用它来解决我们前面列出的广义积分问题时，的确非常好(除较难对付的第五个问题外)，这一点马上就会看到。

广义积分的基本技巧是作变量替换，消去奇异点，或将无穷的积分区间映射到有限的积分区间，例如恒等式

$$\int_a^b f(x)dx = \int_{1/b}^{1/a} \frac{1}{t^2} f\left(\frac{1}{t}\right)dt \qquad ab > 0 \tag{4.4.2}$$

适用于 $b\to\infty$，$a$ 为正数或是 $a\to-\infty$，$b$ 为负数的情况下，任意函数的收敛速度至少如同 $1/x^2$ 的收敛速度。

我们可先作式(4.4.2)中的变量替换，然后用 **qromo** 和 **midpnt** 作数值计算；或者也可以在数值算法中作变量替换。我们倾向于用后一种方法，因为它对用户是透明的。仅对 **midpnt** 作一些修改即可实现式(4.4.2)，称为 **midinf**。$b$ 可为无穷大(准确地说，应是一个很大的数，如 $1\times10^{30}$)，或 $a$ 为负穷大：

```
#define FUNC(x) ((*funk)(1.0/(x))/((x)*(x)))        变量的变换效果

float midinf(float (*funk)(float), float aa, float bb, int n)
    本程序可用于替换 midpnt，即返回值为从 aa 到 bb 对 funk 求积分的第 n 次的精细结果，只是应按1/x 均匀分布的点
    求函数值，而不是按 x。这样上限 bb 可以是计算机允许的正的极大值，或是下限 aa 为负的极小值，但这两种情况不
    能同时出现，aa、bb 符号必须相同。
{
    float x,tnm,sum,del,ddel,b,a;
    static float s;
    int it,j;

    b=1.0/aa;                        这两行改变积分限
    a=1.0/bb;
    if (n == 1) {                    从这里起程序与midpnt完全一样
        return (s=(b-a)*FUNC(0.5*(a+b)));
    } else {
        for(it=1,j=1;j<n-1;j++) it *= 3;
        tnm=it;
        del=(b-a)/(3.0*tnm);
        ddel=del+del;
        x=a+0.5*del;
        sum=0.0;
        for (j=1;j<=it;j++) {
            sum += FUNC(x);
            x += ddel;
            sum += FUNC(x);
            x += del;
        }
        return (s=(s+(b-a)*sum/tnm)/3.0);
    }
}
```

若需从负的下极限到正无穷大上作积分，可在某个正数点处将积分分成两部分来进行，例如：

```
answer=qromo(func, -5.0,2.0,midpnt)+qromo(funk,2.0,1.0e30,midinf);
```

又如何选择分割点呢？应选取一个充分大的正数点，使函数 funk 至少在这点就开始下降，而一直下降到无穷大处时，函数趋近零。第二次调用 **qromo** 的多项式外推是处理 $1/x$ 的多项式，而不是 $x$ 的多项式。

当下限是一个可积的、满足幂规律的极点时，也可作变量替换。若被积函数在 $x=a$ 附近按 $(x-a)^{\gamma}$，$0\leqslant\gamma<1$ 形式发散，则用恒等式

$$\int_a^b f(x)dx = \frac{1}{1-\gamma}\int_0^{(b-a)^{1-\gamma}} t^{\frac{\gamma}{1-\gamma}} f\left(t^{\frac{1}{1-\gamma}}+a\right)dt \qquad (b>a) \tag{4.4.3}$$

若奇异点在上限处，则用恒等式

$$\int_a^b f(x)dx = \frac{1}{1-\gamma}\int_0^{(b-a)^{1-\gamma}} t^{\frac{\gamma}{1-\gamma}} f(b - t^{\frac{1}{1-\gamma}})dt \qquad (b > a) \tag{4.4.4}$$

若上下限都是奇异点，则在中间某处将积分分开，如上面的例子。

若在逆平方根奇异点的情况下，则式(4.4.3)和式(4.4.4)就特别简单，这种情况在实际中经常出现：

$$\int_a^b f(x)dx = \int_0^{\sqrt{b-a}} 2tf(a + t^2)dt \qquad (b > a) \tag{4.4.5}$$

上式是 $a$ 处为奇异点。

而 $b$ 处为奇异点时则

$$\int_a^b f(x)dx = \int_0^{\sqrt{b-a}} 2tf(b - t^2)dt \qquad (b > a) \tag{4.4.6}$$

这里再一次编写对用户透明的替换 **midpnt** 的程序，以实现变量替换，所定义的程序能自动进行变量替换：

```
#include <math.h>

#define FUNC(x) (2.0*(x)*(*funk)(aa+(x)*(x)))

float midsql(float (*funk)(float), float aa, float bb, int n)
    本程序可替换 midpnt，只是被积函数在下限 aa 处可以有一个逆平方根的奇异点。
{
    float x,tnm,sum,del,ddel,a,b;
    static float s;
    int it,j;

    b=sqrt(bb-aa);
    a=0.0;
    if (n == 1) {        程序的其余部分与 midpnt 完全相同，则省略
```

同样有：

```
#include <math.h>

#define FUNC(x) (2.0*(x)*(*funk)(bb-(x)*(x)))

float midsqu(float (*funk)(float), float aa, float bb, int n)
    本程序可替换 midpnt，只是被积函数在上限 bb 处可以有一个逆平方根奇异点。
{
    float x,tnm,sum,del,ddel,a,b;
    static float s;
    int it,j;

    b=sqrt(bb-aa);
    a=0.0;
    if (n == 1) {        程序的其余部分与 midpnt 完全相同，则省略
```

最后举一个例子，它足以说明这些公式是如何导出的。设积分上限为无穷大，被积函数是指数衰减，则需有变量替换，即将 $e^{-x}dx$ 变换为 $(\pm)dt$（选择正负号以使新变量的上限大于下限），可由下面替换来积分：

$$t = e^{-x} \qquad \text{或} \qquad x = -\log t \tag{4.4.7}$$

这样就使

$$\int_{x=a}^{\infty} f(x)dx = \int_{t=0}^{e^{-a}} f(-\log t)\frac{dt}{t} \tag{4.4.8}$$

实现对用户透明的程序为：

```
include <math.h>

define FUNC(x) ((*funk)(-log(x))/(x))

float midexp(float (*funk)(float), float aa, float bb, int n)
   本程序可替换 midpnt，只是 bb 为无穷大(其销使用时不需要)。函数 funk 在无限远处呈指数迅速衰减。
{
   float x,tnm,sum,del,ddel,a,b;
   static float s;
   int it,j;

   b=exp(-aa);
   a=0.0;
   if (n == 1){        程序的其余部分 midpnt 完全相同，则省略
```

## 4.5 高斯求积法与正交多项式

第4.1节中的公式求函数积分，是用一系列等间隔点上的函数值，乘以一些精心选择的系数后求和来近似的。我们看到，只要选择系数时有较多的自由性，就总可以得到越高阶的积分公式。**高斯(Ganssian)求积法**的思想是不仅能给以选择系数的自由，而且还能自由选择不同坐标位置点上的函数值；这些坐标点不再是等间隔的了。这种处理就有了两个自由度，这就导致了高斯(Ganssian)求积分公式的阶次是有同样多个函数值的牛顿-柯特斯(Newton-Cotes)求积公式的阶次的两倍。

听起来让人不能相信吧?但的确如此。其中关键的一点已熟悉得不必再强调了；那就是高阶次并非就意味着高精度。仅当被积函数很平滑时(指在可被多项式很好地逼近的意义下)，阶次高才有高精度。

此外，高斯求积分公式很有用，也是因为它还有另一特性；就是对由多项式乘以某已知函数 $W(x)$ 得到的一类被积函数，可以通过选择权系数和坐标值使求得的积分很精确，而对一般多项式被积函数并不能如此。这时，函数 $W(x)$ 可用来在积分中消去被积函数的可积奇异点。也就是说，给定 $W(x)$ 及整数 $N$，可以找到一系列权 $w_j$ 和一系列坐标 $x_j$，当 $f(x)$ 为多项式时，使得逼近式

$$\int_a^b W(x)f(x)dx \approx \sum_{j=1}^{N} w_j f(x_j) \tag{4.5.1}$$

是精确的，例如求积分

$$\int_{-1}^{1} \frac{\exp(-\cos^2 x)}{\sqrt{1-x^2}}dx \tag{4.5.2}$$

(必须承认，它看上去不是很自然的积分)，我们感兴趣的是，基于下面选择的高斯求积分公式

$$W(x) = \frac{1}{\sqrt{1 - x^2}} \tag{4.5.3}$$

积分区间为$(-1,1)$。(这种变换方法称为**高斯-切比雪夫**积分，这样做的原因读者马上就会明白。)

注意，积分公式(4.5.1)也可用权函数$W(x)$写成并不显而易见的形式：令$g(x) \equiv W(x)f(x)$且$v_j = w_j/W(x_j)$，则式(4.5.1)变成

$$\int_a^b g(x)dx \approx \sum_{j=1}^{N} v_j g(x_j) \tag{4.5.4}$$

函数$W(x)$哪里去了？它正隐含在其中。这里暗示说明，对形如多项式乘以$W(x)$的被积函数能达到较高阶精度，而对平滑性很好的函数却反而达不到高阶精度。我们首先要确定是按式(4.5.1)的形式，还是采用式(4.5.4)的形式来求积分，然后再对给定的$W(x)$求权和坐标点列。

下面例子是$W(x)=1$，$N=10$时的积分程序，其中有已给定的坐标点和权。此时权和坐标点对于积分中心点对称，故实际上只有五个不同值：

```
float qgaus(float (*func)(float), float a, float b)
    返回函数func从a到b的积分，由10点高斯-勒让德积分完成：仅在积分区域10个内点上求函数值。
{
    int j;
    float xr,xm,dx,s;
    static float x[]={0.0,0.1488743389,0.4333953941,          横坐标和权
        0.6794095682,0.8650633666,0.9739065285};               每个数组的第一个值没
    static float w[]={0.0,0.2955242247,0.2692667193,           有用
        0.2190863625,0.1494513491,0.0666713443};

    xm=0.5*(b+a);
    xr=0.5*(b-a);
    s=0;                        将为函数平均值的两倍，因为10个权(上面五个数每个用
    for (j=1;j<=5;j++) {            两次)
        dx=xr*x[j];
        s += w[j]*((*func)(xm+dx)+(*func)(xm-dx));
    }
    return s *= xr;             改变答案的尺度以达到积分的范围
}
```

上例表明，即使不懂高斯求积法理论，也照样可以用这种方法：只要在书上找到列表的权和坐标点(见[1]、[2])。不过这一理论十分精彩，若选用某一特别的$W(x)$需自己构造权和坐标点的表时，则懂一些这方面理论就很方便。因此，这里不作证明地给出一些方便读者的有用结论，结论中假定$W(x)$在$(a,b)$区间内不改变符号，实际中通常也是如此。

高斯求积分理论要追溯到1814年，当时高斯是用连分数来解决这个问题的。1826年雅可比(Jacobi)用正交多项式又推出高斯的结果。用正交多项式对任意权函数系统的处理方法，则主要归功于克里斯托弗尔(Christoffel)在1877年的工作。在介绍正交多项式之前，我们先将感兴趣的积分区间固定为$(a,b)$。

我们定义"两个函数$f$,$g$对权函数$W$的**数积**"为

$$\langle f|g\rangle \equiv \int_a^b W(x)f(x)g(x)dx \tag{4.5.5}$$

数积是一个数，而非$x$的函数。若数积为零，则称两个函数**正交**，若函数与其自身的数积为

1，则称函数是归一的。一组两两正交且分别归一化了的函数的集合称为**正交归一集**。

存在一个多项式集合：(i)对每一个 $j=0,1,2\cdots$，恰好只包含一个 $j$ 次多项式，记为 $p_j(x)$。(ii)对某一特定的权函数，这些多项式两两正交。以下的递推关系是用于寻找这种集合的构造过程。

$$
\begin{aligned}
p_{-1}(x) &\equiv 0 \\
p_0(x) &\equiv 1 \\
p_{j+1}(x) &= (x-a_j)p_j(x) - b_j p_{j-1}(x) \qquad j=0,1,2,\ldots
\end{aligned}
\tag{4.5.6}
$$

其中

$$
\begin{aligned}
a_j &= \frac{\langle xp_j|p_j\rangle}{\langle p_j|p_j\rangle} \qquad j=0,1,\ldots \\
b_j &= \frac{\langle p_j|p_j\rangle}{\langle p_{j-1}|p_{j-1}\rangle} \qquad j=1,2,\ldots
\end{aligned}
\tag{4.5.7}
$$

系数 $b_0$是任意的，我们可以取它为零。

由(4.5.6)得到的多项式为**首一多项式**，即其首项(如 $p_j(x)$的 $x^j$)系数为1。用 $p_j(x)$除以常数 $[\langle p_j|p_j\rangle]^{1/2}$即可得到正交归一的多项式集。也可用其它方法对多项式归一化。要把某已知多项式转换成首一多项式，只要知道 $p_j$ 中 $x^j$ 的系数，设为 $\lambda_j$；然后，用 $\lambda_j$除以 $p_j$ 中各项即可得到首一多项式。注意在递推关系式(4.5.6)中，这个系数依赖于所采用的正交关系。

多项式 $p_j(x)$在$(a,b)$区间内，恰好有 $j$ 个不相等的根，且这些根分别插在 $p_{j-1}(x)$的 $j-1$个根之间，即 $p_{j-1}(x)$的每两个根之间恰好有一个 $p_j(x)$的根。利用此性质可方便找到根：可以 $p_1(x)$的一个根开始，依次将每个 $j$，用牛顿法或其它求根方法精确地找到 $p_j(x)$的根(见第九章)。

为什么要找到正交归一多项式的所有根呢?因为在区间$(a,b)$内，权函数为 $W(x)$的 $N$ 点高斯求积分式为式(4.5.1)和(4.5.4)，它的坐标点恰好正是，同样区间内同样权函数的正交多项式 $p_N(x)$的根。这就是高斯求积法的基本定理，由此可在任何特殊情况下找到横坐标。

找到横坐标点 $x_1,x_2,\cdots,x_N$ 后，就得找权 $w_j$，$j=1,\cdots,N$，其方法之一是解线性方程组

$$
\begin{bmatrix}
p_0(x_1) & \ldots & p_0(x_N) \\
p_1(x_1) & \ldots & p_1(x_N) \\
\vdots & & \vdots \\
p_{N-1}(x_1) & \ldots & p_{N-1}(x_N)
\end{bmatrix}
\begin{bmatrix}
w_1 \\ w_2 \\ \vdots \\ w_N
\end{bmatrix}
=
\begin{bmatrix}
\int_a^b W(x)p_0(x)dx \\ 0 \\ \vdots \\ 0
\end{bmatrix}
\tag{4.5.8}
$$

式(4.5.8)只给出了权，根据式(4.5.1)就能计算出前 $N$ 个正交多项式的积分值。注意式(4.5.8)中右边的零，这是因为 $p_1(x),\cdots,p_{N-1}(x)$都与 $p_0(x)$正交，而 $p_0(x)$为常量。可以看出，由所得到的权，也可求得后面的 $N-1$个多项式的积分。这就是说，对等于小于$2N-1$次的多项式求积分是精确的，另处，还有一种求权的方法(其证明超出本书范围)，是按公式

$$
w_j = \frac{\langle p_{N-1}|p_{N-1}\rangle}{p_{N-1}(x_j)p'_N(x_j)} \tag{4.5.9}
$$

其中 $p'_N(x_j)$为正交多项式在其零点 $x_j$ 处的导数。

可见高斯求积计算法分两步：(i)得到正交多项式 $p_0,\cdots,p_N$，即求式(4.5.6)中系数 $a_i$，$b_i$；(ii)求 $p_N(x)$的零点及其相应的权值。对于一些经典的正交多项式，系数 $a_i$，$b_i$ 已经算出(下面式(4.5.10)～(4.5.14))，故第一步可以省略。但是若碰到"非经典"的权函数 $W(x)$，而且不知道系数 $a_i$，$b_i$，此时构造一个相关正交多项式集就很重要。我们将在本节的最后讨论这个问题。

### 4.5.1 横坐标和权的计算

这件工作难易的程度，取决于读者对权函数及其相关多项式的了解程度。若是很经典、已被仔细研究过的正交多项式，那事情就很明了，包括其零点的良好逼近也是可知的。此时可将其逼近值作为初始猜测值，应用牛顿法就会很快地收敛，牛顿法中需要的导数 $p_N'(x)$，也可以根据 $p_N$ 和 $p_{N-1}$由标准关系式导出。权则由式(4.5.9)可很方便地得到。对于下面谈到的著名示例中，这种直接求根方法比任何其它方法快了3—5倍。

下面列出一些权函数，区间及递推关系式，由这些可以求得最常用的正交多项式及其相应的高斯求积分公式。

高斯-勒让德：(Gauss-Legendre)

$$\begin{gathered} W(x) = 1 \qquad -1 < x < 1 \\ (j+1)P_{j+1} = (2j+1)xP_j - jP_{j-1} \end{gathered} \tag{4.5.10}$$

高斯-切比雪夫(Gauss-Chebyshev)

$$\begin{gathered} W(x) = (1-x^2)^{-1/2} \qquad -1 < x < 1 \\ T_{j+1} = 2xT_j - T_{j-1} \end{gathered} \tag{4.5.11}$$

高斯-拉盖尔(Gauss-Laguerre)

$$\begin{gathered} W(x) = x^\alpha e^{-x} \qquad 0 < x < \infty \\ (j+1)L_{j+1}^\alpha = (-x+2j+\alpha+1)L_j^\alpha - (j+\alpha)L_{j-1}^\alpha \end{gathered} \tag{4.5.12}$$

高斯-埃尔米特(Gauss-Hermite)

$$\begin{gathered} W(x) = e^{-x^2} \qquad -\infty < x < \infty \\ H_{j+1} = 2xH_j - 2jH_{j-1} \end{gathered} \tag{4.5.13}$$

高斯-雅可比(Guass-Jacobi)

$$\begin{gathered} W(x) = (1-x)^\alpha(1+x)^\beta \qquad -1 < x < 1 \\ c_jP_{j+1}^{(\alpha,\beta)} = (d_j+e_jx)P_j^{(\alpha,\beta)} - f_jP_{j-1}^{(\alpha,\beta)} \end{gathered} \tag{4.5.14}$$

其中系数 $c_j$,$d_j$,$e_j$,$f_j$ 为

$$\begin{aligned} c_j &= 2(j+1)(j+\alpha+\beta+1)(2j+\alpha+\beta) \\ d_j &= (2j+\alpha+\beta+1)(\alpha^2-\beta^2) \\ e_j &= (2j+\alpha+\beta)(2j+\alpha+\beta+1)(2j+\alpha+\beta+2) \\ f_j &= 2(j+\alpha)(j+\beta)(2j+\alpha+\beta+2) \end{aligned} \tag{4.5.15}$$

现在，我们分别给出每种情况下计算横坐标点和权的程序。首先是，最常用的高斯—勒让德横坐标和权集。本程序由 G. B. Rybicki 设计，式(4.5.9)在高斯-勒让德情况下的特殊形式为

$$w_i = \frac{2}{(1-x_i^2)[P_N'(x_i)]^2} \tag{4.5.16}$$

在程序中，还将积分区间$(x_1, x_2)$映射为$(-1,1)$，输出结果为高斯求积公式需要的坐标点$x_i$和$w_i$，高斯积分公式为：

$$\int_{x_1}^{x_2} f(x)dx = \sum_{i=1}^{N} w_i f(x_i) \qquad (4.[illegible]7)$$

```
#include <math.h>
#define EPS 3.0e-11          EPS为相对精度

void gaulen(float x1, float x2, float x[], float w[], int n)
   给定积分上下限x1,x2及给定n，本程序返回长度为n的数组x[1..n]及w[1..n]，其中分别存放坐标点及n点高斯
   勒让德求积公式的权值。
{
    int m,j,i;
    double z1,z,xm,xl,pp,p3,p2,p1;                      高精度对本程序是个好主意

    m=(n+1)/2;                                          区间中根是对称的，这样只需求其中的一半
    xm=0.5*(x2+x1);
    xl=0.5*(x2-x1);
    for (i=1;i<=m;i++) {                                对求的根作循环
        z=cos(3.141592654*(i-0.25)/(n+0.5));            从上面对第i个根的逼近值开始，进入用牛顿法作精
                                                        细划分的主循环
        do {
            p1=1.0;
            p2=0.0;
            for (j=1;j<=n;j++) {                        从递推关系求得勒让德多项式在z点的值
                p3=p2;
                p2=p1;
                p1=((2.0*j-1.0)*z*p2-(j-1.0)*p3)/j;     现在p1就是所求勒让德多项式，下面将
            }                                           由它与p2的标准关系式，计算其导数
            pp=n*(z*p1-p2)/(z*z-1.0);                   pp，p2是比p1低一阶的多项式
            z1=z;
            z=z1-p1/pp;                                 牛顿法
        } while (fabs(z-z1) > EPS);
        x[i]=xm-xl*z;                                   将根值映射到指定区间
        x[n+1-i]=xm+xl*z;                               再求其对称的关联点
        w[i]=2.0*xl/((1.0-z*z)*pp*pp);                  计算权及其对称关联值
        w[n+1-i]=w[i];
    }
}
```

下面的三个程序是根据Stroud和Secrest[2]给出的根作为初始逼近值。第一个是高斯—拉盖尔的坐标点和权值，能应用于下面的积分公式：

$$\int_0^{\infty} x^{\alpha} e^{-x} f(x)dx = \sum_{j=1}^{N} w_j f(x_j) \qquad ([illegible]8)$$

```
#include <math.h>
#define EPS 3.0e-14          若用户没有该精度则增加EPS
#define MAXIT 10

void gaulag(float x[], float w[], int n, float alf)
   给定α(即广义拉盖尔多项式的参数)，本程序返回数组x[1..n]及w[1..n]，其中分别存放n点高斯-拉盖尔求积公式
   的横坐标点和权。最小坐标为x[1]，最大为x[n]。
```

```
float gammln(float xx);
void nrerror(char error_text[]);
int i,its,j;
float ai;
double p1,p2,p3,pp,z,z1;                                  高精度对本程序是个好主意

for (i=1;i<=n;i++) {                                     在求各根上循环
    if (i == 1) {                                        最小根的初始预测值
        z=(1.0+alf)*(3.0+0.92*alf)/(1.0+2.4*n+1.8*alf);
    } else if (i == 2) {                                 第二个根的初始预测值
        z += (15.0+6.25*alf)/(1.0+0.9*alf+2.5*n);
    } else {                                             其它根的初始预测值
        ai=i-2;
        z += ((1.0+2.55*ai)/(1.9*ai)+1.26*ai*alf/
            (1.0+3.5*ai))*(z-x[i-2])/(1.0+0.3*alf);
    }
    for (its=1;its<=MAXIT;its++) {                       牛顿法做细分
        p1=1.0;
        p2=0.0;
        for (j=1;j<=n;j++) {                             从递推关系求得拉盖尔多项式
            p3=p2;                                       在z点的值
            p2=p1;
            p1=((2*j-1+alf-z)*p2-(j-1+alf)*p3)/j;    现在p1就是所求拉盖尔多项式，下面就
        }                                            将由它与p2的标准关系式计算其导数
        pp=(n*p1-(n+alf)*p2)/z;                      pp。p2是比p1低一阶的多项式
        z1=z;
        z=z1-p1/pp;                                      牛顿公式
        if (fabs(z-z1) <= EPS) break;
    }
    if (its > MAXIT) nrerror("too many iterations in gaulag");
    x[i]=z;                                              存储根和权
    w[i] = -exp(gammln(alf+n)-gammln((float)n))/(pp*n*p2);
}
}
```

下面的程序计算的是高斯-埃尔米特坐标和权。若我们使用函数的标准规一化值，如式(4.5.13)，我们发现当$N$很大时计算会溢出，这是因为会产生不同的阶乘，为避免产生这种情况，可以使用多项式$\tilde{H}_j$的首一化集，此集合可由下面递推式产生：

$$\tilde{H}_{-1}=0,\quad \tilde{H}_0=\frac{1}{\pi^{1/4}},\quad \tilde{H}_{j+1}=x\sqrt{\frac{2}{j+1}}\tilde{H}_j-\sqrt{\frac{j}{j+1}}\tilde{H}_{j-1} \qquad (4.5.19)$$

求权的公式变成

$$w_j=\frac{2}{(\tilde{H}_j')^2} \qquad (4.5.20)$$

求规一化导数的公式为

$$\tilde{H}_j'=\sqrt{2j}\,\tilde{H}_{j-1} \qquad (4.5.21)$$

由程序**gauher**返回的坐标点和权可用于积分公式

$$\int_{-\infty}^{\infty} e^{-x^2}f(x)dx=\sum_{j=1}^{N}w_jf(x_j) \qquad (4.5.22)$$

```
#include <math.h>
#define EPS 3.0e-14                                      相对精度
```

```
#define PIM4 0.7511255444692        1/π^{1/4}
#define MAXIT 10                    最多迭代次数

void gauher (float x[], float w[], int n)
    给定n,本程序返回数组x[1…n]和w[1…n],其中分别存放n点高斯-埃尔米特积分公式的坐标值和权值。最大坐
    标是x[1],最小的是x[n]
{
    void nrerror(char error_text[]);
    int i,its,j,m;
    double p1,p2,p3,pp,z,z1;                          高精度对本程序是一个好主意
    m=(n+1)/2;                                        根对原点对称,所以只求它一半
    for (i=1;i<=m;i++) {                              对所求的根循环
        if (i == 1) {                                 最大根的初始猜测值
            z=sqrt((doulbe)(2*n+1))-1.85575*pow((double)(2*n+1), -0.16667);
        } else if (i == 2) {                          第二大根的初始猜测值
            z -= 1.14*pow((double)n,0.426)/z;
        } else if (i == 3) {                          第三大根的初始猜测值
            z=1.86*z-0.86*x[1];
        } else if (i == 4) {                          第四大根的初始猜测值
            z=1.91*z-0.91*x[2];
        } else {                                      其它根的起始猜测值
            z=2.0*z-x[i-2];
        }
        for (its=1;its<=MAXIT;its++) {                由牛顿法做细分
            p1=PIM4;
            p2=0.0;
            for (j=1;j<=n;j++) {                      由递推式循环求得埃尔米特多项式在z处的值
                p3=p2;
                p2=p1;
                p1=z*sqrt(2.0/j)*p2-sqrt(((double)(j-1))/j)*p3;   p1就是所求埃尔米特多项式。
                                                      下面将利用比p1低一阶的多项式p2
            pp=sqrt((double)2*n)*p2;                  的关系式(4.5.12)求得其导数pp
            z1=z;                                     牛顿公式
            z=z1-p1/pp;
            if (fabs(z-z1) <= EPS) break;
        }
        if (its > MAXIT) nrerror("too many iterations in gauher");
        x[i]=z;                                       存根
        x[n+1-i] = -z;                                及其对称关联根
        w[i]=2.0/(pp*pp);                             计算权
        w[n+1-i]=w[i];                                及其对称关联值
    }
}
```

最后,给出计算高斯-雅可比坐标点和权的程序。这些坐标值和权值用于下面积分公式

$$\int_{-1}^{1}(1-x)^{\alpha}(1+x)^{\beta}f(x)dx=\sum_{j=1}^{N}w_j f(x_j) \tag{4.5.23}$$

```
#include <math.h>
#define EPS 3.0e-14           若没有此精度,则增加EPS
#define MAXIT 10

void gaujac(float x[], float w[], int n, float alf, float bet)
    给定alf及bet,即雅可比(Jacobi)多项式的参数α和β,本程序返回数组x[1…n]和w[1…n],其中分别存放n点高
    斯-雅可比积分公式的坐标值和权值。最大坐标为x[1],最小坐标为x[n]。
```

```
float gammln(float xx);
void nrerror(char error_text[]);
int i,its,j;
float alfbet,an,bn,r1,r2,r3;
double a,b,c,p1,p2,p3,pp,temp,z,z1;                  高精度对本程序是一个好主意

for (i=1;i<=n;i++) {                                  对所求的根循环
    if (i == 1) {                                     最大根的初始猜测值
        an=alf/n;
        bn=bet/n;
        r1=(1.0+alf)*(2.78/(4.0+n*n)+0.768*an/n);
        r2=1.0+1.48*an+0.96*bn+0.452*an*an+0.83*an*bn;
        z=1.0-r1/r2;
    } else if (i == 2) {                              对第二个大的根的初始猜测值
        r1=(4.1+alf)/((1.0+alf)*(1.0+0.156*alf));
        r2=1.0+0.06*(n-8.0)*(1.0+0.12*alf)/n;
        r3=1.0+0.012*bet*(1.0+0.25*fabs(alf))/n;
        z -= (1.0-z)*r1*r2*r3;
    } else if (i == 3) {                              对第三个大根的初始猜测值
        r1=(1.67+0.28*alf)/(1.0+0.37*alf);
        r2=1.0+0.22*(n-8.0)/n;
        r3=1.0+8.0*bet/((6.28+bet)*n*n);
        z -= (x[1]-z)*r1*r2*r3;
    } else if (i == n-1) {                            次最小根的初始猜测值
        r1=(1.0+.235*bet)/(0.766+0.119*bet);
        r2=1.0/(1.0+0.639*(n-4.0)/(1.0+0.71*(n-4.0)));
        r3=1.0/(1.0+20.0*alf/((7.5+alf)*n*n));
        z += (z-x[n-3])*r1*r2*r3;
    } else if (i == n) {                              最小根的初始猜测值
        r1=(1.0+0.37*bet)/(1.67+0.28*bet);
        r2=1.0/(1.0+0.22*(n-8.0)/n);
        r3=1.0/(1.0+8.0*alf/((6.28+alf)*n*n));
        z += (z-x[n-2])*r1*r2*r3;
    } else {                                          其它根的初始猜测值
        z=3.0*x[i-1]-3.0*x[i-2]+x[i-3];
    }
    alfbet=alf+bet;
    for (its=1;its<=MAXIT;its++) {                   由牛顿法做细分
        temp=2.0+alfbet;                              从p0,p1开始递推以避免α+β=0或-1时零作除数
        p1=(alf-bet+temp*z)/2.0;
        p2=1.0;
        for (j=2;j<=n;j++) {                          由递推式循环求得在z点处的雅可比多项式的值
            p3=p2;
            p2=p1;
            temp=2*j+alfbet;
            a=2*j*(j+alfbet)*(temp-2.0);
            b=(temp-1.0)*(alf*alf-bet*bet+temp*(temp-2.0)*z);
            c=2.0*(j-1+alf)*(j-1+bet)*temp;
            p1=(b*p2-c*p3)/a                          p1就是所求雅可比多项式。下面将由其与p2的标准关系
        }                                             式求其导数pp,p2是比p1低一阶的多项式
        pp=(n*(alf-bet-temp*z)*p1+2.0*(n+alf)*(n+bet)*p2)/(temp*(1.0-z*z));
        z1=z;
        z=z1-p1/pp;                                   牛顿公式
        if (fabs(z-z1) <= EPS) break;
    }
    if (its > MAXIT) nrerror("too many iterations in gaujac");
    x[i]=z;                                           存储根及权
    w[i]=exp(gammln(alf+n)+gammln(bet+n)-gammln(n+1.0)-
        gammln(n+alfbet+1.0))*temp*pow(2.0,alfbet)/(pp*p2);
```

勒让德多项式是雅比可多项式当 $\alpha=\beta=0$ 时的特例，不过最好还是对每种情况编写一个程序，例如，上面给出的程序 **gauleg**。而切比雪夫多项式对应于 $\alpha=\beta=-1/2$（见第5.8节），可以解析地写出其坐标点和权为：

$$x_j = \cos\left(\frac{\pi(j-\frac{1}{2})}{N}\right), \qquad w_j = \frac{\pi}{N} \tag{4.5.24}$$

## 4.5.2 递推式已知时的情况

现在讨论一下，不知道函数正交多项式零点的最佳初始估计值，但可用系数 $a_j$ 和 $b_j$ 求得它们的情况。$p_N(x)$ 的零点自然就是 $N$ 点高斯求积分公式中的坐标值。求权最有效方法是用公式(4.5.9)，因为一般情况下，其中导数 $p'_N$ 很容易由式(4.5.6)微分得到，对于某些经典多项式则可由其特殊关系式得到。注意，式(4.5.9)仅对写成为首一多项式的形式才有效。对其它整理形式，则需另有一个因子 $\lambda_N/\lambda_{N-1}$，$\lambda_N$ 为 $p_N$ 中 $x^N$ 的系数。

除了某些已讨论过的特殊情况外，求坐标点的最好方法并不是用象牛顿法之类的求根法。一般更快的方法有 Galub–Welsch[] 算法，此方法基于 Wilf[] 工作。这个算法就是，将项 $xp_j$ 加到(4.5.6)式左边，而将 $p_j$ 项加到右边。递推关系式可写成矩阵形式：

$$x\begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_{N-2} \\ p_{N-1} \end{bmatrix} = \begin{bmatrix} a_0 & 1 & & & \\ b_1 & a_1 & 1 & & \\ & \vdots & \vdots & & \\ & & b_{N-2} & a_{N-2} & 1 \\ & & & b_{N-1} & a_{N-1} \end{bmatrix} \cdot \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_{N-2} \\ p_{N-1} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ p_N \end{bmatrix}$$

或是

$$x\mathbf{p} = \mathbf{T}\cdot\mathbf{p} + p_N\mathbf{e}_{N-1} \tag{4.5.25}$$

其中 $\mathbf{T}$ 为三对角矩阵；$\mathbf{p}$ 为 $p_0, p_1, \cdots, p_{N-1}$ 的列向量；$\mathbf{e}_{N-1}$ 为单位向量，即其第 $N-1$ 位（最后）为1，其余为零。$\mathbf{T}$ 矩阵可由对角相似转换矩阵 $\mathbf{D}$ 变成对称形式：

$$\mathbf{J} = \mathbf{DTD}^{-1} = \begin{bmatrix} a_0 & \sqrt{b_1} & & & \\ \sqrt{b_1} & a_1 & \sqrt{b_2} & & \\ & \vdots & \vdots & & \\ & & \sqrt{b_{N-2}} & a_{N-2} & \sqrt{b_{N-1}} \\ & & & \sqrt{b_{N-1}} & a_{N-1} \end{bmatrix} \tag{4.5.26}$$

$\mathbf{J}$ 矩阵称为**雅可比矩阵**（注意不要与其它有相同名字，但出现在完全不同问题中的矩阵混淆了）。从式(4.5.25)可以看出，$p_N(x_j)=0$ 等价于 $x_j$ 为 $\mathbf{T}$ 的特征值。因为相似转换后特征值不变，可见 $x_j$ 是对称三对角矩阵 $\mathbf{J}$ 的特征值。此外，Wilf[] 还证明了，若 $\mathbf{v}_j$ 为对应于特征值 $x_j$ 的特征向量，经归一化使得 $\mathbf{v}\cdot\mathbf{v}=1$，则

$$w_j = \mu_0 v_{j,1}^2 \tag{4.5.27}$$

其中

$$\mu_0 = \int_a^b W(x)dx \tag{4.5.28}$$

上式中 $v_{j,1}$ 是 v 的第一分量。在第十一章中我们会看到，寻找一个对称三对角矩阵的特征值和特征向量并不是件困难的事。这里给出一个程序，可由系数 $a_j$ 和 $b_j$ 找到坐标点和权值。记住，若正交多项式的递推式未整理成首一形式，则通过乘以一个数 $\lambda_j$，可以很容易转化成首一多项式的形式。

```
#include <math.h>
#include "nrutil.h"

void gaucof(int n,float a[],float b[], float amu0, float x[], float w[])
```

由 Jacobi 矩阵计算高斯求积分公式的坐标点和权值。输入是 a[1…n] 及 b[1…n]，分别为正交多项式集合的递推关系式系数，量 $\mu_0 = \int_a^b W(x)dx$ 作为 amu0的输入。返回坐标点 x[1…n]按递减顺序，相应的权为 w[1…n]。数组 a，b 被修改。若修改程序 **tqli** 和 **eigsrt**，只计算每个特征向量的第一个分量，则执行速度可以加快。

```
{
    void eigsrt(float d[], float **v, int n);
    ovid tqli(float d[], float e[], int n, float **z);
    int i,j;
    float **z;

    z=matrix(1,n,1,n);
    for (i=1;i<=n;i++) {
        if (i != 1) b[i]=sqrt(b[i]);                       设置雅可比矩阵的上对角
        for (j=1;j<=n;j++) z[i][j]=(float)(i == j);        设置单位矩阵，由 tqli 计算特征向量
    }
    tqli(a,b,n,z);
    eigsrt(a,z,n);                                          按递减顺序排放特征值
    for (i=1;i<=n;i++) {
        x[i]=a[i];
        w[i]=amu0*z[1][i]*z[1][i];                          式(4.5.12)
    }
    free_matrix(z,1,n,1,n);
}
```

## 4.5.3 具有非经典权的正交多项式

本节有点特殊，讨论若权函数不是上面处理的某些经典形式，而且也不知道 **gaucof** 中式(4.5.6)的递推式的 $a_j$、$b_j$ 值。此时，需要另外一种求 $a_j$、$b_j$ 的方法。

**Stieltjes 方法**由式(4.5.7)算得 $a_0$，由式(4.5.6)求得 $p_1(x)$，已知 $p_0$、$p_1$后，则可由式(4.5.7)求得 $a_1$、$b_1$，如此递推下去。但是怎么计算式(4.5.7)中的内积呢？

一种方法是，将每个 $p_j(x)$ 解析地表示为 $x$ 的多项式形式，然后逐项相乘求内积。当知道权函数的前 $2N$ 个矩时，这种方法似乎是可行的，此矩为

$$\mu_j = \int_a^b x^j W(x)dx \qquad j = 0,1,2,\ldots,2N-1 \tag{4.5.29}$$

然而，根据 $\mu_j$ 的值求解系数 $a_j$、$b_j$ 的代数方程，得到的结果呈现严重的病态。即使是双精度变量，当 $N=12$ 时也很不精确。所以不能使用基于式(4.5.29)矩的方法。

Sack 和 Donovan[5]发现，如果表示 $p_j$ 时，不用 $x$ 的幂作为基本函数集，而是使用另外一组正交多项式如 $\pi_j(x)$，则数值的稳定性会大大地改善。粗略地讲，稳定性改善是因为当求内积积分时，多项式作为"幂"采样$(a,b)$区间比幂作为基要好，特别是当权函数象 $W(x)$时。

假设已知修正矩值

$$\nu_j = \int_a^b \pi_j(x)W(x)dx \qquad j = 0,1,2,\ldots,2N-1 \tag{4.5.30}$$

其中 $\pi_j$ 满足类似式(4.5.6)的递推关系式

$$\pi_{-1}(x) \equiv 0$$
$$\pi_0(x) \equiv 1 \qquad (4.5.31)$$
$$\pi_{j+1}(x) = (x - \alpha_j)\pi_j(x) - \beta_j \pi_{j-1}(x) \qquad j = 0,1,2,\ldots,$$

系数 $\alpha_j$、$\beta_j$ 是确切已知的。Wheeler[8]找到一个 $O(N^2)$ 有效算法，与 Sack 和 Donovan 算法是等效的，通过一组中间过渡量求 $a_j$、$b_j$

$$\sigma_{k,l} = \langle p_k | \pi_l \rangle \qquad k,l \geqslant -1 \qquad (4.5.32)$$

初始化

$$\sigma_{-1,l} = 0 \qquad l = 1,2,\ldots,2N-2$$
$$\sigma_{0,l} = \nu_l \qquad l = 0,1,\ldots,2N-1$$
$$a_0 = \alpha_0 + \frac{\nu_1}{\nu_0} \qquad (4.5.33)$$
$$b_0 = 0$$

则对于 $k = 1,2,\cdots,N-1$，计算

$$\sigma_{k,l} = \sigma_{k-1,l+1} + (\alpha_l - a_{k-1})\sigma_{k-1,l} - b_{k-1}\sigma_{k-2,l} + \beta_l \sigma_{k-1,l-1}$$
$$l = k,k+1,\ldots,2N-k-1$$
$$a_k = \alpha_k - \frac{\sigma_{k-1,k}}{\sigma_{k-1,k-1}} + \frac{\sigma_{k,k+1}}{\sigma_{k,k}} \qquad (4.5.34)$$
$$b_k = \frac{\sigma_{k,k}}{\sigma_{k-1,k-1}}$$

注意，需要的话归一化因子也很容易算出

$$\langle p_0 | p_0 \rangle = \nu_0$$
$$\langle p_j | p_j \rangle = b_j \langle p_{j-1} | p_{j-1} \rangle \qquad j = 1,2,\ldots \qquad (4.5.35)$$

参考文献[7]中有此算法的推理。

Wheeler 算法需要计算式(4.5.30)的修正矩值。实际情况中通常为闭型的形式，或可使用递推关系式。对有限区间$(a,b)$，此算法非常有效；对无限区间，该算法并未能完全消除病态性。Gautschi[illegible]提出用变换变量将无限区间转化为一个有限区间，然后再用一个合适的离散化过程计算其内积。详情可见参考文献。

下面给出由 Wheeler 算法产生系数 $a_j$、$b_j$ 的程序 **orthog**，给定系数 $\alpha_j$、$\beta_j$ 及修正矩 $\nu_j$。为保持与程序 **gaucof** 的兼容性，向量 $\alpha$、$\beta$、$a$、$b$ 为单位偏移向量，相应地，将 $\sigma$ 矩阵的索引号增为2，即 sig[k,l] $= \sigma_{k-2,l-2}$。

```
#include "nrutil.h"

void orthog(int n, float anu[], float alpha[], float beta[], float a[], float b[])
```

由 Wheeler 算法，计算权函数为 $W(x)$ 的正交多项式的递推关系式中的系数 $a_j$,$b_j$,$j=0,1,\cdots,N-1$。输入数组 alpha[1..2*n−1]和 beta[1..2*n−1]为选定基为正交多项式的递推关系式中的系数 $\alpha_j$,$\beta_j$,$j=0,\cdots,2N-2$。输入修正矩为 anu[1..2*n]。前 $n$ 个系数返回数组 a[1..n]和 b[1..n]中。

```
{
    int k,l;
    float **sig;
    int looptmp;

    sig=matrix(1,2*n+1,1,2*n+1);
    looptmp=2*n;
    for (l=3;l<=looptmp;l++) sig[1][l]=0.0;        初始化(4.5.33)式
    looptmp++;
    for (l=2;l<=looptmp;l++) sig[2][l]=anu[l-1];
    a[1]=alpha[1]+anu[2]/anu[1];
```

```
    b[1]=0.0;
    for (k=3;k<=n+1;k++) {                                  (4.5.34)式
        looptmp=2*n-k+3;
        for (l=k;l<=looptmp;l++) {
            sig[k][l]=sig[k-1][l+1]+(alpha[l-1]-a[k-2])*sig[k-1][l]-
                b[k-2]*sig[k-2][l]+beta[l-1]*sig[k-1][l-1];
        }
        a[k-1]=alpha[k-1]+sig[k][k+1]/sig[k][k]-sig[k-1][k]/sig[k-1][k-1];
        b[k-1]=sig[k][k]/sig[k-1][k-1];
    }
    free_matrix(sig,1,2*n+1,1,2*n+1);
}
```

作为应用 orthog 的示例是，考察如何在区间(0,1)上，产生权函数 $W(x)=-\log x$ 的正交多项式。一个合适的 $\pi_j$ 集是为移位勒让德多项式。

$$\pi_j = \frac{(j!)^2}{(2j)!}P_j(2x-1) \tag{4.5.36}$$

$P_j$ 前面的因子使多项式为首一的。递推关系式(4.5.31)中的系数为

$$\begin{aligned} \alpha_j &= \frac{1}{2} & j &= 0,1,\dots \\ \beta_j &= \frac{1}{4(4-j^{-2})} & j &= 1,2,\dots \end{aligned} \tag{4.5.37}$$

修正矩为

$$\nu_j = \begin{cases} 1 & j=0 \\ \dfrac{(-1)^j(j!)^2}{j(j+1)(2j)!} & j\geqslant 1 \end{cases} \tag{4.5.38}$$

用这些作为输入去调用 orthog，即可在 $N$ 很大时得到满足机器精度的多项式。然后用此权函数就可以计算高斯求积分公式了。这个看起来似乎很简单的问题，然而在 Sack 和 Donovan 之前，其实是很困难的。

### 4.5.4 高斯积分推广

推广高斯求积法的思想有多种途径。很重要的一种推广是**预设节点**：即先指定某些必须包含在坐标集中的点，并且选择权及其余坐标点，使得积分法尽可能准确。最常见的例子是 Gauss-Radau 积分，其节点之一必须为区间的端点，即 $a$ 或 $b$。而 Gauss-Lobatto 积分则必须 $a$，$b$ 都为节点。Golub[16]为这些情况设计了与 **gaucof** 类似的算法。

第二种重要的推广是 Gauss-Kronrod 公式。对一般高斯求积分公式，当 $N$ 增加时，坐标集中没有相同的点。这样在用加大 $N$ 的方法估计积分误差时，不能使用以前的函数值。Kronrod[11]提出寻找最优序列的方法，每次都能使用前一次的所有坐标值。若从 $N=m$ 开始，新增加 $n$ 个点，则会有$2n+m$ 个自由变量：$n$ 个新坐标点，$n$ 个权值和对应于以前固定坐标点的 $m$ 个新的权值，所以期望能达到的最大精确度为$2n+m-1$。现在问题是在实际中，当要求的所有坐标点都在$(a,b)$中时，这个最大准确度能达到吗?这一问题的答案现在还不清楚。

Kronord 发现当 $n=m+1$时，可以得到高斯—勒让德积分的最佳推广。Patterson[12]找到如何去连续计算这种推广的方法。在自动求积分程序[13]中，求一个函数的积分并能达到所指定的精度要求，通常使用的序列是 $N=10,21,43,87,\cdots$。

参考文献和进一步读物：

Abramowitz, M. and Stegun, I. A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, vol. 55 § 25.4. [1]

Stroud, A. H. and Secrest, D. 1966, *Gaussian Quadrature Formulas* (Englewood Cliffs, NJ: Prentice Hall). [2]

Golub, G. H. and Welsch, J. H. 1969, *Mathematics of Computation*, vol. 23, pp. 221～230 and A1～A10. [3]

Wilf, H. S. 1962, *Mathematics for the Physical Sciences* (New York: Wiley), Problem 9, p. 80. [4]

Sack, R. A. and Donovan, A. F. 1971/72, *Numerische Mathematik*, vol. 18, pp. 465～478. [5]

Wheeler, J. C. 1974, *Rocky Mountain Journal of Mathematics*, vol. 4, pp. 287～296. [6]

Gautschi, W. 1987, in *Recent Advances in Numerical Analysis*, C. de Boor and G. H. Golub, eds. (New York: Academic Press), pp. 45～72. [7]

Gautschi, W. 1981, in *E. B. Christoffel*, P. L. Butzer and F. Feher, eds (Basel: Birkhauser Verlag), pp. 72～147. [8]

Gautschi, W. 1990, in *Orthogonal Polynomials*, P. Nevai, ed. (Dordrecht: Kluwer Academic Publishers), pp. 181～216. [9]

Golub, G. H. 1973, *SIAM Review*, vol. 15, pp. 318～334. [10]

Kronrod, A. S. 1964, *Doklady Akademii Nauk SSSR*, vol. 154, pp. 283～286 (in Russian). [11]

Patterson, T. N. L. 1968, *Mathematics of Computation*, vol. 22, pp. 847～856 and C1～C11; 1969, op. cit., vol. 23, p. 892. [12]

Piessens, R., de Doncker, E., Uberhuber, C. W., and Kahaner, D. K. 1983, *QUADPACK: A Subroutine Package for Automatic Integration* (New York: Springer-Verlag). [13]

## 4.6 多维积分

在大于一维的区域上，多个变量的函数的积分不太容易计算，有两个原因：第一，$N$ 维空间求函数值的点数，将按照一维情况下的 $N$ 次幂增长。粗略地说，求一个一维积分需求30个点的函数值，则三维情况下达到同样精度水平，就需要 30000 次求函数值。第二，$N$ 维积分区域须由 $N-1$维的边界来定义，而边界本身极其复杂：例如，不一定就是凸的或是连通的。而一维积分边界则只有两个数，上限和下限。

碰到多维积分问题时，首先要问的问题是，“能解析地将维数降低吗？”例如，称为单变量函数 $f(t)$ 的**多重积分**可由下面公式降为一维积分：

$$
\begin{aligned}
&\int_0^x dt_n \int_0^{t_n} dt_{n-1} \cdots \int_0^{t_3} dt_2 \int_0^{t_2} f(t_1)dt_1 \\
&= \frac{1}{(n-1)!}\int_0^x (x-t)^{n-1} f(t)dt
\end{aligned}
\tag{4.6.1}
$$

或者函数有某种依赖独立变量的特殊对称性质，若边界也有这种对称性，则维数可降低。例如，三维空间，中球对称函数在球形域中的积分，就可以在极坐标下降为一维积分。

对下面提出的问题有两种完全不同的解决办法，问题是：积分区域边界的形状是简单的，还是复杂的？该区域内，被积函数是简单平滑的，还是复杂的，或者是局部有显著峰值的？要求精度高不高，要求结果的精度是准确到百分之一，还是百分之几？

若边界复杂，被积函数在很小区域内峰值不显著，且允许相对较低的精度，则可用蒙特卡罗(**Monte Carlo**)**积分方法**。这种方法编程简单，特别是对粗略的形式。首先，须知道一个

区域，其边界是包含积分区域的简单的边界；还须有一种算法，能判别随机点是落在积分区域的内部还是外部。蒙特卡罗方法先求随机点处的函数值，然后由这些随机值来估算积分。第七章将讨论更详细的细节及较复杂的方法。

若边界简单，函数很平滑，则用残留逼近，将多维积分分成多重一维积分，或者多重高斯求积分，这样做效果好而且速度比较快[1]。若精度要求较高，则无论如何这种逼近方法是唯一的选择。因为蒙特卡罗方法本质上渐近地收敛速度较慢。

精度要求不高时，若被积函数变化较慢，在积分区域内又很平滑，则用多重一维积分。若被积函数振荡或不连续，但在小区域内峰值不显著时，则用蒙特卡罗方法。

若被积函数在小区域内峰值显著，并且已知这些区域，则可将积分分成在几个区域内分别进行，使其在每个区域内都连续。若这些区域不可知，那就没有什么办法了(在本节程度范围内)。不能指望积分程序会搜索到 $N$ 维空间内某种对积分影响较大的未知小块(但可参考第7.8节)。

读了上面的内容，读者可能想了解多重一维积分的逼近方法。下面谈谈其原理。我们的考察限定在 $x$、$y$、$z$ 空间的三维积分。二维或高于三维的情况完全类似。

第一步，确定积分区域：(i) $x$ 的上下限，记为 $x_1$，$x_2$；(ii)在确定的 $x$ 值处 $y$ 的上下限，记 $y_1(x)$，$y_2(x)$；(iii)在确定的 $x$，$y$ 处 $z$ 的上下限，记为 $z_1(x,y)$，$z_2(x,y)$。换言之，找到 $x_1$，$x_2$ 值及函数 $y_1(x)$，$y_2(x)$，$z_1(xy)$ 和 $z_2(x,y)$ 使得

$$
\begin{aligned}
I &= \iiint dxdydzf(x,y,z) \\
&= \int_{x_1}^{x_2} dx \int_{y_1(x)}^{y_2(x)} dy \int_{z_1(x,y)}^{z_2(x,y)} dzf(x,y,z)
\end{aligned}
\tag{4.6.2}
$$

例如，单位圆上的二维积分为

$$
\int_{-1}^{1} dx \int_{-\sqrt{1-x^2}}^{\sqrt{1-x^2}} dyf(x,y) \tag{4.6.3}
$$

现将最内层积分定义为 $G(x,y)$

$$
G(x,y) \equiv \int_{z_1(x,y)}^{z_2(x,y)} f(x,y,z)dz \tag{4.6.4}
$$

对 $G(x,y)$ 的积分记为函数 $H(x)$

$$
H(x) \equiv \int_{y_1(x)}^{y_2(x)} G(x,y)dy \tag{4.6.5}
$$

最后，对 $H(x)$ 的积分即为所求：

$$
I = \int_{x_1}^{x_2} H(x)dx \tag{4.6.6}
$$

可递归地调用一些基本的一维积分程序，来实现方程式(4.6.4)～(4.6.6)(如下面程序中的 **qgaus**)：求外层积分 $I$ 只须调一次，求中间积分则需调用多次，求内层积分调用次数就更多了(见图4.6.1)。

当前的 $x$，$y$ 值和指向函数 func 的指针，通过静态的顶端变量传递给中间的每次调用。

```
static float xsav, ysav;
static float (*nrfunc)(float, float, float);
```

float qrad3d(float (*func)(float, float, float), float x1, float x2)

程序返回三维区域上函数func的积分，而三维区域由x1、x2的值和式(4.6.2)所定义的函数yy1、yy2、z1、z2确定，这些函数由用户提供，(这里函数y1、y2写为yy1、yy2，以免与某些C函数库中贝塞尔函数名字混淆)，积分由递归地调用qgaus求得

```
{
    float qgaus(float (*func)(float), float a, float b);
    float f1(float x);

    nrfunc=func;
    return qgaus(f1,x1,x2);
}

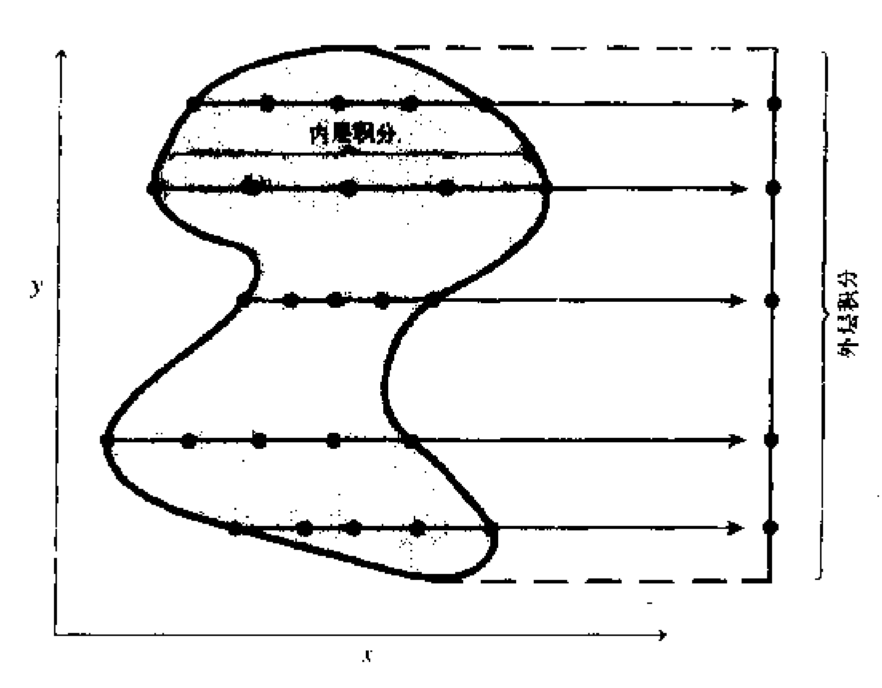
float f1(float x)                     这是(4.6.5)式的H函数
{
    float qgaus(float (*func)(float), float a, float b);
    float f2(float y);
    float yy1(float),yy2(float);

    xsav=x;
    return qgaus(f2,yy1(x),yy2(x));
}

float f2(float y)                     这是(4.6.4)式的G函数
{
    float qgaus(float (*func)(float), float a, float b);
    float f3(float z);
    float z1(float,float),z2(float,float);

    ysav=y;
    return qgaus(f3,z1(xsav,y),z2(xsav,y));
}

float f3(float z)                     x、y处被积函数f(x、y、z).
{
    return (*nrfunc)(xsav,ysav,z);
}
```




外层对y的积分请求内层对x沿选定y的网线上的积分。内层积分程序求函数在x处的值。这样求函数值一般比笛卡儿网点法精确

图4.6.1 不规则区域上进行二维积分时计算函数值

用户必须提供的函数如以下范例：

```
float func(float x, float y, float z);          待积分的三维函数
float yy1(float x);
float yy2(float x);
float z1(float x,float y);
float z2(float x,float y);
```

参考文献和进一步读物：


Stroud, A. H1971, *Approximate Calculation of Multiple Integrals* (Englewood Cliffs, NJ: Prentice Hall). [1]

# 第五章 函数求值

## 5.0 引言

函数求值方法很多，本章讨论如何选择一种合适的方法。第六章将通过给出一些特殊函数的子程序，来解释并应用这些方法。故本章和下一章目的是一致的，不过还是有些区别：本章阐述得最清晰明了的一般方法，下一章一些奇特的特殊函数却不一定选用。比较这两章，我们会看到所谓"一般"与"特殊"方法之间的协调关系。

本章将讨论一般方法及如何编写子程序来估计函数值，这些函数也许是"特殊"的，但还没有特殊到包括至第六章或标准程序库中。

## 5.1 级数和其收敛性

大家知道，解析函数可在某点 $x_0$ 的邻域内展开成级数：

$$f(x) = \sum_{k=0}^{\infty} a_k (x - x_0)^k \tag{5.1.1}$$

用这个级数可直接估值。当然没有必要每次都计算出 $x-x_0$ 的 $k$ 次幂，只需记下 $k-1$ 次幂再作一次乘法。同样，系数 $a$ 的形式通常也可利用前面的工作：$k!$ 或 $(2k)!$ 项都可以用一次或两次乘法求得。

如何知道加项已足够多了呢？实际应用时，级数的高次项最好能快速减小，不然级数方法就不是最好的方法。通常做法是，当新增加的一项的数值小于前面和的 $\varepsilon$（一个小正数）倍时，就不再加项。这样做，有时数学上是不太严格的。（注意，可能有 $a_k=0$ 的极端情况。）

级数表示的缺点之一是，在复平面上远离 $x_0$ 处遇到奇异性，此幂级数保证不收敛。不过这也不太要紧：书上找的级数（或自己设计的），一般可知其收敛半径。然而处处收敛（数学意义上）的级数有一个严重问题，就是在任何处收敛得都不够快，不能用数值计算方法。sine 函数和第一类 Bessel 函数就是两个熟悉的例子：

$$\sin(x) = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} x^{(2k+1)} \tag{5.1.2}$$

$$J_n(x) = \left(\frac{x}{2}\right)^n \sum_{k=0}^{\infty} \frac{\left(-\frac{1}{4}x^2\right)^k}{k!(n+k)!} \tag{5.1.3}$$

[illegible]都要到 $k \gg |x|$ 时才开始收敛，在此之前，项值一直增

[illegible]价值。

[illegible]和的序列收敛）。对于像式(5.1.2)和(5.

2.3)那种通项需一直增加的级数，这些方法作用不大。而对于通项的模是递减的级数，某些加速收敛的效果是惊人地好。**Aitken $\delta^2$过程**是一个近似几何收敛的、对级数部分和式进行外推的简单公式。设 $S_{n-1}, S_n, S_{n+1}$ 为三个相邻的部分和，更好的估计是

$$S'_n \equiv S_{n+1} - \frac{(S_{n+1} - S_n)^2}{S_{n+1} - 2S_n + S_{n-1}} \tag{5.1.4}$$

使用式(5.1.4)时，对任意整数 $p$ 可用 $n+p$ 和 $n-p$ 分别替换 $n+1$ 和 $n-1$，得到 $S'_n$ 序列后，再次使用式(5.1.4)，如此类推下去。(实际上，经过第一次迭代后已大部分收敛，以后迭代收敛已很少)。注意，式(5.1.4)必须按所给式计算。代数上，有一种等价的对舍入误差更敏感的形式。

对**交错级数**(指和式中项的符号交替变化)，**Euler 变换**是有力的工具。通常它首先计算少量 $n-1$ 项的和，然后对从第 $n$ 项开始级数的其余部分作变换，公式($n$ 为偶数)为：

$$\sum_{s=0}^{\infty}(-1)^s u_s = u_0 - u_1 + u_2 \ldots - u_{n-1} + \sum_{s=0}^{\infty}\frac{(-1)^s}{2^{s+1}}[\Delta^s u_n] \tag{5.1.5}$$

这里 $\Delta$ 是**前向差分算子**，例如：

$$\begin{aligned}
\Delta u_n &\equiv u_{n+1} - u_n \\
\Delta^2 u_n &\equiv u_{n+2} - 2u_{n+1} + u_n \\
\Delta^3 u_n &\equiv u_{n+3} - 3u_{n+2} + 3u_{n+1} - u_n
\end{aligned} \tag{5.1.6}$$

自然不必算出式(5.1.5)右边无穷项之和，可只计算前 $p$ 项和，用式(5.1.6)从 $u_n$ 开始得到前 $p$ 个差分。

欧拉变换不仅可用于交错级数，有时某个形式上发散的级数，仅根据前几项就可得到精确的求和结果。它也被大量用于渐近级数求和。比较明智的方法是，只计算到该项的值不再增加为止，同时做一些独立的数值检验工作，确保得到的结果是有意义的。

范·维金加登(Van Wijngaarden)发现欧拉变换的一个精巧的实现方式，即每次按顺序合并原交错级数的一项。每次合并时，或是将 $p$ 增1，即再做一次高次差分；或是例行地将 $n$ 加1，此时不必在原 $n$ 值上，再重做一遍所有的差分运算。根据使级数收敛最快来决定是增加 $n$ 还是增加 $p$。Van Wijngaarden 方法仅需一个向量来存放部分差分。算法如下：

```
#include <math.h>

void eulsum(float *sum, float term, int jterm, float wksp[])
    和式中合并第 jterm 项，其值为 term，sum 是输入即先前计算出的部分和，并是新的部分和的输出。首次调用此子程
    序，令 jterm=1，输入级数首项。第二次调用时，term 置为级数第二项，其符号与首次调用时相反，令 jterm=2，依此
    类推。wksp 是调用程序所提供的工作空间的数组，其维数至少要大于需合并项的最大数。
{
    int j;
    static int nterm;
    float tmp,dum;

    if (jterm == 1) {                          初始化：
        nterm=1;                               wksp中存放差值数
        *sum=0.5*(wksp[1]=term);               返回初次估计
    } else {
        tmp=wksp[1];
        wksp[1]=term;
        for (j=1;j<=nterm-1;j++) {             用van Wijngaarden算法更新当前存放值
```

```
            dum=wksp[j+1];
             wksp[j+1]=0.5*(wksp[j]+tmp);
             tmp=dum;
        }
        wksp[nterm+1]=0.5*(wksp[nterm]+tmp);
        if (fabs(wksp[nterm+1]) <= fabs(wksp[nterm]))          增加 p
            *sum += (0.5*wksp[++nterm]);           表加长
        else                                       增加 n
            *sum += wksp[nterm+1];                 表不加长
    }
}
```

这种欧拉技巧很有效，但不能直接用于正项级数。有时将正项级数转换成交错级数，再用欧拉变换。为此，Van Wijngaarden 找到一个转换公式

$$\sum_{r=1}^{\infty} v_r = \sum_{r=1}^{\infty} (-1)^{r-1} w_r \tag{5.1.7}$$

其中

$$w_r \equiv v_r + 2v_{2r} + 4v_{4r} + 8v_{8r} + \cdots \tag{5.1.8}$$

式(5.1.7)和(5.1.8)用二维和代替一维和，式(5.1.7)中每一项是式(5.1.8)的一个无穷和。这种方法能节省工作量，看上去比较奇怪！这是由于式(5.1.8)式下标指数按2的幂增长很快，仅需很少几项就能收敛得很精确，只要对"随机"数 $r$ 有效地算出 $u_r$。对于 $r$ 这种标准的"最新"诀窍，上述由式(5.1.1)导出的方法，已不能再使用。

事实上，欧拉变换是一个更为一般的级数变换的特例。设某已知函数 $g(z)$ 的级数

$$g(z) = \sum_{n=0}^{\infty} b_n z^n \tag{5.1.9}$$

现在对另一未知级数求和

$$f(z) = \sum_{n=0}^{\infty} c_n b_n z^n \tag{5.1.10}$$

不难看出(见[2])式(5.1.10)可写为

$$f(z) = \sum_{n=0}^{\infty} [\Delta^{(n)} c_0] \frac{g^{(n)}}{n!} z^n \tag{5.1.11}$$

通常它收敛要快得多。其中的 $\Delta^{(n)} c_0$ 是 $n$ 阶有限差分算子式(5.1.6)，$\Delta^{(0)} c_0 \equiv c_0$，$g^{(n)}$ 则为 $g(z)$ 的 $n$ 阶导数。

用 $g(z)$ 替代，即可得到通常的欧拉变换方程(5.1.5)并 $n=0$，例如

$$g(z) = \frac{1}{1+z} = 1 - z + z^2 - z^3 - \cdots \tag{5.1.12}$$

代入式(5.1.11)，并令 $z=1$。

有时虽然计算效果不好，也还得用级数来计算函数。例如，我们可以利用所得到的数值逼近为函数的近似形式，它在后面将要用到(参见第5.8节)。在计算缓慢收敛级数的许多项之和时，应注意舍入误差！浮点数表示时，按顺序求一列数之和，从最小项开始比从最大项开始的结果更为准确。将项两两合成一对，再两两合成一组的效果更好，这样在加法中就包含了比较大小的操作。

参考文献和进一步读物：

Goodwin, E. T. (ed) 1961, *Modern Computing Methods*, 2nd ed. (New York: Philosophical Library).

Chapter 13 [van Wijngaarden's transformations]. [1]

Mathews, J., and Walker, R.L. 1970, *Mathematical Methods of Physics*, 2nd ed. (Reading, MA: W. A. Benjamin/Addison-Wesley), §2.3. [2]

## 5.2 连分式求值

通常,连分式可作为科学计算中求函数值的有效工具。

连分式如下面形式:

$$f(x) = b_0 + \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3 + \cfrac{a_4}{b_4 + \cfrac{a_5}{b_5 + \cdots}}}}} \tag{5.2.1}$$

印刷时写成:

$$f(x) = b_0 + \frac{a_1}{b_1 +}\frac{a_2}{b_2 +}\frac{a_3}{b_3 +}\frac{a_4}{b_4 +}\frac{a_5}{b_5 +}\cdots \tag{5.2.2}$$

式(5.2.1)和(5.2.2)中 $a_i$,$b_i$ 可以是 $x$ 的函数,一般为线性的,或最坏的为 $x$ 的二次单项式(如 $x$ 或 $x^2$ 的常数倍)。例如,正切函数连分式表示为:

$$\tan(x) = \frac{x}{1-}\frac{x^2}{3-}\frac{x^2}{5-}\frac{x^2}{7-}\cdots \tag{5.2.3}$$

连分式通常比级数展开收敛要快得多,并且收敛区域更大(但不是总是包括级数的收敛域)。有时级数收敛最坏的地方却连分式收敛最好,虽然不总是这样的。Blanch[1]给出了有关连分式最有用的收敛测试的详细介绍。

有一些标准技术,包括重要的商—差算法在内,都不外乎是连分式逼近,还是幂级数逼近和有理函数逼近。有关这方面内容参见 Acton[2],更详细的讨论见 Fike[3]。

当计算连分式何时能停止呢?跟级数不同,不能从左到右估计式(5.2.1),当变化很小时就停下来。而写成式(5.2.1)的形式后,计算连分式唯一办法是从右到左,首先估计(盲目!)从哪里开始。然而,正确方法不是如此。

正确的方法是,应用一个联系连分式和有理数逼近的结果,并给出一个从左到右估计式(5.2.1)或(5.2.2)的方法。设 $f_n$ 表示由 $a_n$ 和 $b_n$ 估计式(5.2.2)的结果,那么

$$f_n = \frac{A_n}{B_n} \tag{5.2.4}$$

其中 $A_n$,$B_n$ 由下列递推式给出:

$$\begin{aligned} A_{-1} &\equiv 0 \qquad B_{-1} \equiv 0 \\ A_0 &\equiv b_0 \qquad B_0 \equiv 1 \\ A_j &= b_jA_{j-1} + a_jA_{j-2} \qquad B_j = b_jB_{j-1} + a_jB_{j-2} \qquad j = 1,2,\ldots,n \end{aligned} \tag{5.2.5}$$

这方法是在1655年由 J. Wallis 发明,并讨论在他的 Arithmetica Infinitorum 一书中[4]。读者能很容易地证明它。

实际上,这算法有某些乏味的性质。由式(5.2.5)经常会递推出一些非常大或非常小的

部分分子 $A_j$ 和分母 $B_j$，这样用浮点数表示时，可能有上溢和下溢的危险。然而递推式(5.2.5)对 $A_j$ 和 $B_j$ 是线性的，因此每次迭代时，都可重新整定当前存放的递推结果，例如，可将 $A_j$、$B_j$、$A_{j-1}$ 和 $B_{j-1}$ 除以 $B_j$，使 $A_j=f_j$。还可供判断是否还需递推下去：只须看最新迭代出的 $f_j$ 和 $f_{j-1}$ 是不是如所期望那样接近。(若有时 $B_j$ 恰好为零，则跳出本次循环，重新归一化。更好的优化方法是，仅当溢出即将发生时才重新整定，以省去不必要的除法，但这样会使算法复杂化)。

有两种新的求连分式的方法。斯蒂特(Steed)的方法不用 $A_j$、$B_j$，而只用了比值 $D_j=B_{j-1}/B_j$。只要递推地用下式计算 $D_j$ 和 $\Delta f_j=f_j-f_{j-1}$

$$D_j = 1/(b_j + a_j D_{j-1}) \tag{5.2.6}$$

$$\Delta f_j = (b_j D_j - 1)\Delta f_{j-1} \tag{5.2.7}$$

斯蒂特法[5]避免了重新调整当前结果。但对某些连分数可能会碰到这种情况，即(5.2.6)式中的分母几乎为零，$D_j$、$\Delta f_j$ 就很大。下一个 $\Delta f_{j+1}$ 虽会抵消这种变化，但在计算 $f_j$ 时数值和时使精度降低。此时程序运行不尽人意，所以，一般斯蒂特方法只在予先知道分母不会很小的情况下才推荐使用。在第6.7节 **bessik** 程序中，此方法将用于某个特殊目的。

求连分式最一般的方法大概要算改进的 Lentz 方法[6]。利用下面两个比值就可以不必对当前结果整定：

$$C_j = A_j/A_{j-1} \qquad D_j = B_{j-1}/B_j \tag{5.2.8}$$

再由下式计算 $f_j$

$$f_j = f_{j-1} C_j D_j \tag{5.2.9}$$

由式(5.2.5)知，显然这两比值满足递推关系式：

$$D_j = 1/(b_j + a_j D_{j-1}) \qquad C_j = b_j + a_j/C_{j-1} \tag{5.2.10}$$

此算法中，$D_j$ 表达式的分母或 $C_j$ 本身有趋近零的危险，这样式(5.2.10)就没有用了。对此 Thompson 和 Barnett[5] 提出修正 Lentz 算法来解决这个问题：它将为零的项用一个很小的量代替，如 $10^{-30}$。只需用这种方法试着将算法运行一个循环，就会发现 $f_1$ 可精确地计算出来。

下面详细列出改进的 Lentz 算法：

- 令 $f_0=b_0$；若 $b_0=0$，则令 $f_0=tiny$
- 令 $C_0=f_0$
- 令 $D_0=0$
- 对 $j=1,2,\cdots$

    令 $D_j=b_j+a_jD_{j-1}$

    若 $D_j=0$，令 $D_j=tiny$

    令 $C_j=b_j+a_j/C_{j-1}$

    若 $C_j=0$，令 $C_j=tiny$

    令 $D_j=1/D_j$

    令 $\Delta_j=C_jD_j$

    令 $f_j=f_{j-1}\Delta_j$

    若 $|\Delta_j-1|<eps$ 则停止

其中 $eps$ 为浮点精度值，可以是 $10^{-7}$ 或 $10^{-15}$。参数 $tiny$ 必须小于 $eps|b_j|$，如 $10^{-30}$。

上面算法是假定 $|f_j - f_{j-1}|$ 足够小时，可以结束连分式求值。通常情况下正是如此，但不能保证总是这样。Jones[7] 为判断求各种连分数值的结束条件，证明了一系列定理。

Lintz 算法目前尚无误差扩散的一般分析，不过经验表明此算法至少不比其它算法差。

## 5.2.1 连分式处理

利用连分式的性质可将连分数写成另外的形式，以加速数值运算过程。一个等价变换是：

$$a_n \to \lambda a_n, \quad b_n \to \lambda b_n, \quad a_{n+1} \to \lambda a_{n+1} \tag{5.2.11}$$

它不会改变连分式的值。选择合适的修整因子 $\lambda$ 可以简化 $a$ 和 $b$ 的形式。当然还可以作连续的等价变换，连分式的相邻连续项可用不同的 $\lambda$ 值。

连分式的奇部分和偶部分也是连分式，其分别各自连续收敛到 $f_{2n}$ 和 $f_{2n+1}$。这里所利用的，主要是因为它们收敛比原来连分式快两倍。如果它们各部分中的项并不比原分式复杂多少，则可以节省很多计算。(5.2.5)偶部分公式为：

$$f_{even} = d_0 + \frac{c_1}{d_1 +}\frac{c_2}{d_2 +}\cdots \tag{5.2.12}$$

其中根据当前变量得

$$\alpha_1 = \frac{a_1}{b_1}$$
$$\alpha_n = \frac{a_n}{b_n b_{n-1}}, \qquad n \geqslant 2 \tag{5.2.13}$$

我们有

$$d_0 = b_0, \quad c_1 = \alpha_1, \quad d_1 = 1 + \alpha_2$$
$$c_n = -\alpha_{2n-1}\alpha_{2n-2}, \qquad d_n = 1 + \alpha_{2n-1} + \alpha_{2n}, \qquad n \geqslant 2 \tag{5.2.14}$$

在 Blanch[1] 著作中可以找到奇部分的类似公式。通常联合利用(5.2.11)式和(5.2.14)式是最佳数值计算方法。

下一章，我们将要经常运用连分式。


参考文献和进一步读物：

Blanch, G. 1964, *SIAM Review*, vol. 6, pp. 383～421. [1]

Acton, F. S. 1970, *Numerical Methods That Work*, 1990, corrected edition (Washington: Mathematical Association of America), Chapter 11. [2]

Fike, C. T. 1968, *Computer Evaluation of Mathematical Functions* (Englewood Cliffs, NJ: Prentice-Hall), §§8.2, 10.4, and 10.5. [3]

Wallis, J. 1695, in *Opera Mathematica*, vol. 1, p. 355, Oxoniae e Theatro Shedoniano. Reprinted by Georg Olms Verlag, Hildesheim, New York (1972). [4]

Thompson, I. J., and Barnett, A. R. 1986, *Journal of Computational Physics*, vol. 64, pp. [illegible]. [5]

Lentz, W. J. 1976, *Applied Optics*, vol. 15, pp. 668～671. [6]

Jones, W. B. 1973, in *Padé Approximants and Their Applications*, P. R. Graves-Morris, ed. (London: Academic Press), p. [illegible]. [7]

## 5.3 多项式和有理函数

$N$ 次多项式在数值上，可用一个存储的系数数组 c[$j$]，$j=0,\cdots,N$ 来表示。其中 c[0] 为多项式中常数项，c[$N$] 为 $x^N$ 系数，当然其它习惯表示法也是可以的。多项式有两种处理方法，一是**数值处理**(如求值)，此时应给定自变量的数值，另一种方法是**代数处理**，只是以某种方式变换系数数组，而不需要某个特定的自变量。首先讨论数值处理方法。

首先，假定读者已知道永远不能用如下方式求多项式的值：

```
p=c[0]+c[1]*x+c[2]*x*x+c[3]*x*x*x+c[4]*x*x*x*x
```

或者(更糟糕!)

```
p=c[0]+c[1]*x+c[2]*pow(x,2.0)+c[3]*pow(x,3.0)+c[4]*pow(x,4.0);
```

计算机革命已经来临，还犯这种错误的人真是不应该!至于是写成

```
p=c[0]+x*(c[1]+x*(c[2]+x*(c[3]+x*c[4])));
```

还是写成

```
p=(((c[4]*x+c[3])*x+c[2])*x+c[1])*x+c[0];
```

就只是习惯问题了。

若系数 c[0…n] 数量很多，程序应为：

```
p=c[n]
for(j=n-1;j>=0;j--) p=p*x+c[j];
```

或

```
p=c[j=n]
while  (j>0)  p=p*x+c[--j];
```

另外，还有一个同时计算多项式 $P(x)$ 及其导数 $dP(x)/dx$ 之值的方法：

```
p=c[n];
dp=0.0;
for(j=n-1;j>=0;j--) {dp=dp*x+p; p=p*x+c[j];}
```

或

```
p=c[j=n];
dp=0.0;
while  (j>0)  {dp=dp*x+p; p=p*x+c[--j];}
```

程序结果是计算多项式值 $p$ 及其导数值 $dp$。

以上方法基本上为**综合除法**，可以推广到一般地同时求多项式及其 $n$ 阶导数值。

```
void ddpoly(float c[], int nc, float x, float pd[],int nd)
    给定 nc 次多项式以数组 c[0..nc]表示的 nc+1 个系数，c[0]为常数项，x 值给定，给定 nd>1，程序在 pd[0]中返回
    多项式在 x 处的值，pd[1]…pd[nd]中返回各阶导数。
{
    int nnd,j,i;
    float cnst=1.0;

    pd[0]=c[nc];
    for (j=1;j<=nd;j++)   pd[j]=0.0;
```

```
for (i=nc-1;i>=0;i--) {
    nnd=(nd < (nc-i) ? nd : nc-i);
    for (j=nnd;j>=1;j--)
        pd[j]=pd[j]*x+pd[j-1];
    pd[0]=pd[0]*x+c[i];
}
for (i=2;i<=nd;i++) {            一阶导数后,出现阶乘因子
    cnst *= i;
    pd[i] *= cnst;
}
```

读者可能奇怪并想知道，$n>3$ 次多项式却用少于 $n$ 次的乘法就能求得结果，为此，须先算出一些辅助系数，并再做些加法。例如多项式

$$P(x)=a_0+a_1x+a_2x^2+a_3x^3+a_4x^4 \tag{5.3.1}$$

可由3次乘法5次加法算出。其中 $a_4>0$，

$$P(x)=[(Ax+B)^2+Ax+C][(Ax+B)^2+D]+E \tag{5.3.2}$$

这里 $A$、$B$、$C$、$D$、$E$ 由下式给出：

$$\begin{aligned} A &= (a_4)^{1/4} \\ B &= \frac{a_3-A^3}{4A^3} \\ D &= 3B^2+8B^3+\frac{a_1A-2a_2B}{A^2} \\ C &= \frac{a_2}{A^2}-2B-6B^2-D \\ E &= a_0-B^4-B^2(C+D)-CD \end{aligned} \tag{5.3.3}$$

五次多项式可由4次乘法，5次加法得出，六次多项式可由4次乘法，7次加法求出。若感兴趣的话，可参阅[3]～[5]。这个主题虽说不很实用，但也是很有意思的，它带有一点第2.11节中快速矩阵乘法的味道。

现在讨论代数处理。一个 $n-1$ 次多项式（范围在[0..n-1]的数组）乘一个多项式因子 $x-a$，可用下面一段程序：

```
c[n]=c[n-1];
for (j=n-1;j>=1;j--) c[j]=c[j-1]-c[j]*a;
c[0] *= (-a);
```

同样地，$n$ 次多项式除以多项式因子 $x-a$（又是综合除法）可用下述程序：

```
rem=c[n];
c[n]=0.0;
for(i=n-1;i>=0;i--){
    swap=c[i];
    c[i]=rem;
    rem=swap+rem*a;
}
```

最后得到一个新的多项式和一个数值余数 rem。

两个一般多项式相乘，每次从每个多项式取一个系数相乘，然后直接求和。两个一般多

项式相除，可以用笔和纸笨拙地勉强算出来，但使用不同优化算法影响却很大。注意，下面的程序基于参考书[3]中的算法。

```
void poldiv(float u[],int n, float v[], int nv, float q[], float r[])
给定n次多项式的n+1个系数u[0..n],及另一个nv次多项式的nv+1个系数v[0..nv],用多项式v去除多项式u
("u"/"v"),商的多项式系数返回到q[0..n]中,余数多项式系数返回到r[0..n]中,数组元素r[nv..n]和q[n-nv
+1..n]返回零值。
{
	int k,j;

	for (j=0;j<=n;j++) {
		r[j]=u[j];
		q[j]=0.0;
	}
	for (k=n-nv;k>=0;k--) {
		q[k]=r[nv+k]/v[nv];
		for (j=nv+k-1;j>=k;j--) r[j] -= q[k]*v[j-k];
	}
	for (j=nv;j<=n;j++) r[j]=0.0;
}
```

## 5.3.1 有理函数

求一个如下所示有理函数之值

$$R(x)=\frac{P_\mu(x)}{Q_\nu(x)}=\frac{p_0+p_1x+\dots+p_\mu x^\mu}{q_0+q_1x+\dots+q_\nu x^\nu} \tag{5.3.4}$$

显然，只须分别求两个多项式值后再相除。通常为方便起见，选择 $q_0=1$，只需分母分子上下同除以 $q_0$ 即可，此外可将两组系数存入一个数组，调用下面求值的标准函数：

```
double ratval(double x, double cof[], int mm, int kk)
给定mm,kk和cof[0..mm+kk],计算并返回函数(cof[0]+cof[1]x+…+cof[mm]x^mm)/(1+cof[mm+1]x+…
+cof[mm+kk]x^kk)。
{
	int j;
	double sumd,sumn;              注意精度!若需要可改变成浮点数

	for (sumn=cof[mm],j=mm-1;j>=0;j--) sumn=sumn*x+cof[j];
	for (sumd=0.0,j=mm+kk;j>=mm+1;j--) sumd=(sumd+cof[j])*x;
	return sumn/(1.0+sumd);
}
```

参考文献和进一步读物：


Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), pp. 183, 190. [1]

Mathews, J., and Walker, R.L. 1970, *Mathematical Methods of Physics*, 2nd ed. (Reading, MA: W.A. Benjamin/Addison-Wesley), pp. 361~363. [2]

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §4.6. [3]

Winograd, S. 1970, *Communications on Pure and Applied Mathematics*, vol. 23, pp. 165~179. [4]

Kronsjö, L. 1987, *Algorithms: Their Complexity and Efficiency*, 2nd ed. (New York: Wiley). [5]

## 5.4 复数运算

正如第1.2节中指出，C 中没有复数运算，这一点对数值计算工作的确很不利。即使在FORTRAN一类有复数数据类型的高级语言中，有时也会遇到，虽复数操作数和结果都可很好地表示，但运算过程中出现上溢和下溢之类令人很麻烦的事。这种情况之所以发生，大概是因为软件公司把实现复数运算这类他们认为很细小的事，分派给经验不足的程序员去完成的缘故。

事实上，复数运算一点也不是件细小的事。加、减自然很容易做，只须对运算数的实部和虚部分别作运算。乘法也不难，只要做4次乘法、一次加法和一次减法，

$$(a+ib)(c+id)=(ac-bd)+i(bc+ad) \tag{5.4.1}$$

（符号 $i$ 前的加法不能算，它只是显式地分开实部和虚部）。但有时如下乘起来更快

$$(a+ib)(c+id)=(ac-bd)+i[(a+b)(c+d)-ac-bd] \tag{5.4.2}$$

它只有三次乘法（$ac$，$bd$，$(a+b)(c+d)$），加上两次加法和三次减法。总的操作次数增加了两次，但是许多机器上乘法是很慢的。

的确，有时候式(5.4.1)和(5.4.2)的最后结果是可表示的，而中间过程却发生溢出，这种情况仅会在最后结果勉强可表示时才会产生。求复数模时，不要按下式计算

$$|a+ib|=\sqrt{a^2+b^2} \qquad (\text{不好!}) \tag{5.4.3}$$

当 $a$ 或 $b$ 大到最大可表示数的平方根时（如等于$10^{19}$，其平方根为$10^{38}$），中间结果会溢出。正确的方法是

$$|a+ib|=\begin{cases}|a|\sqrt{1+(b/a)^2} & |a|\geqslant|b|\\ |b|\sqrt{1+(a/b)^2} & |a|<|b|\end{cases} \tag{5.4.4}$$

复数作除法时，应用类似的方法以避免上溢、下溢或失去精度，

$$\frac{a+ib}{c+id}=\begin{cases}\dfrac{[a+b(d/c)]+i[b-a(d/c)]}{c+d(d/c)} & |c|\geqslant|d|\\[2ex] \dfrac{[a(c/d)+b]+i[b(c/d)-a]}{c(c/d)+d} & |c|<|d|\end{cases} \tag{5.4.5}$$

当然其中的子表达式，如 $c/d$，$d/c$，只需要计算一次。

复数平方根就更复杂了，因为除了必须监视中间结果外，还得讨论选择求得的结果（发生到负实轴上）。求 $c+id$ 的平方根，先计算

$$w\equiv\begin{cases}0 & c=d=0\\ \sqrt{|c|}\sqrt{\dfrac{1+\sqrt{1+(d/c)^2}}{2}} & |c|\geqslant|d|\\ \sqrt{|d|}\sqrt{\dfrac{|c/d|+\sqrt{1+(c/d)^2}}{2}} & |c|<|d|\end{cases} \tag{5.4.6}$$

求得结果为

$$
\sqrt{c+id} = \begin{cases} 0 & w = 0 \\ w + i\left(\dfrac{d}{2w}\right) & w \neq 0,\ c \geqslant 0 \\ \dfrac{|d|}{2w} + iw & w \neq 0,\ c < 0,\ d \geqslant 0 \\ \dfrac{|d|}{2w} - iw & w \neq 0,\ c < 0,\ d < 0 \end{cases} \tag{5.4.7}
$$

实现这些算法的程序列于附录C中。

## 5.5 递推关系及克伦肖递推公式

许多有用的函数满足递推关系，如：

$$(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x) \tag{5.5.1}$$

$$J_{n+1}(x) = \frac{2n}{x}J_n(x) - J_{n-1}(x) \tag{5.5.2}$$

$$nE_{n+1}(x) = e^{-x} - xE_n(x) \tag{5.5.3}$$

$$\cos n\theta = 2\cos\theta\cos(n-1)\theta - \cos(n-2)\theta \tag{5.5.4}$$

$$\sin n\theta = 2\cos\theta\sin(n-1)\theta - \sin(n-2)\theta \tag{5.5.5}$$

前三个函数分别为勒让德多项式，第一类贝塞尔函数和指数积分[1]。当从两个相邻 $n$ 项的值往大或小的方向扩展计算序列项时，这些关系式很有用。

这里关于三角函数式(5.5.4)和(5.5.5)插几句话。如果说程序运行的大部分时间总是在计算三角函数，那一定是什么地方有问题。对于自变量为线性序列 $\theta=\theta_0+n\delta, n=0,1,2\cdots$ 的三角函数，使用下面递推关系很有效：

$$
\begin{aligned}
\cos(\theta+\delta) &= \cos\theta - [\alpha\cos\theta + \beta\sin\theta] \\
\sin(\theta+\delta) &= \sin\theta - [\alpha\sin\theta - \beta\cos\theta]
\end{aligned} \tag{5.5.6}
$$

其中 $\alpha,\beta$ 是预先计算的系数

$$\alpha \equiv 2\sin^2\left(\frac{\delta}{2}\right) \qquad \beta \equiv \sin\delta \tag{5.5.7}$$

不用通常方法(等价地)求角度而采用这种方法的原因是，当增量 $\delta$ 很小时，$\alpha,\beta$ 仍在起作用。同样，式(5.5.6)中加法应按方括号中的顺序求和。我们在第12章中，处理傅立叶变换时，将反复利用式(5.5.6)。

有时，另外一种诀窍也很有用，就是注意到 $\sin\theta$ 和 $\cos\theta$ 都可以通过 tan 计算

$$t \equiv \tan\left(\frac{\theta}{2}\right) \quad \cos\theta = \frac{1-t^2}{1+t^2} \quad \sin\theta = \frac{2t}{1+t^2} \tag{5.5.8}$$

当求 $\sin\theta$ 和 $\cos\theta$ 时都必须求一次 tan，另外还有两次乘法，2次除法及2次加法。当在机器上求三角函数比较慢时，这样做能节省计算。但当 $\theta\to\pm\pi$ 时，要求特殊处理。另外也应注意，现在许多机器上求三角函数也很快，可上机试一试才能知道式(5.5.8)是否更快些

### 5.5.1 递推式的稳定性

在递推方向上($n$ 增加或减小)，递推式对于舍入误差并不总是稳定的。一个三项线性递推关系式

$$y_{n+1} + a_n y_n + b_n y_{n-1} = 0 \qquad n = 1, 2, \ldots \tag{5.5.9}$$

有两组线性独立的根 $f_n$,$g_n$。其中只有一组根 $f_n$ 的序列才是我们要求的。另一组 $g_n$ 可能沿指定方向是指数增长,或指数下降,或指数中性(如按某种幂规律增长或下降)。若为指数增长,则在指定方向上此递推式没有实际用处。例如式(5.5.2)在 $n$ 增加方向上,当 $x<n$ 时就是这种情况。故在 $n$ 很大时,不能用式(5.5.2)前向递推出贝塞尔函数。

严格一点说,若 $\qquad f_n/g_n \to 0 \quad$ 当 $\quad n\to\infty \qquad$ (5.5.10)

则 $f_n$ 称为递推关系式(5.5.9)的**最小解**。非最小解如 $g_n$ 称为**强解**。最小解若存在,则是唯一的;强解则不然——例如可以在 $g_n$ 上增加 $f_n$ 的任意倍数。利用前向递推可以求任意强解值,但不是**最小解**(遗憾的是最小解恰恰就是我们要求的解。)

阿布拉莫维茨(Abramowitz)和斯特根(Stegun)[1]给出了一列在 n 增大或减小方向上都稳定的递推关系式。当然不能包括所有可能的公式。给定某个函数 $f_n(x)$的递推关系,读者花五分钟就可以自己检验一下:在感兴趣范围内取一个固定 $x$,不要从 $f_j(x)$ 和 $f_{j-1}(x)$ 的实际值开始递推,而是对它们分别赋 1,0 值,然后再分别赋 0,1 值。对此两种不同的初始情况,在递推方向上(从 $j$ 开始增大或减小),作出递归序列 10 至 20 项。注意,两个序列对应项之差,若差值小于允许值(如绝对值小于10),则递推是稳定的。若增长缓慢,递推可能轻微不稳定但可以忍受。若快速增长,则存在指数增长的解。如果已知所求函数是与增长的解对应,则可保留递推公式,例如 $n$ 增长时的贝塞尔函数,参见第6.5节。如果不知道这个解是所求函数对应的解,此时应舍弃递推公式。应该注意,为计算两起始函数 $f_j(x)$和 $f_{j-1}(x)$而寻找数值方法之前,应先作这些检验工作;稳定性是递推关系本身的属性,与初始值无关。

还有一种很有启发性的检验稳定性的方法,就是把具有常系数的线性递推关系代替原有的递推关系,例如(5.5.2)式可写成:

$$y_{n+1} - 2\gamma y_n + y_{n-1} = 0 \qquad (5.5.11)$$

其中 $\gamma \equiv \frac{n}{x}$ 视为常数。设解的形式为 $y=a^n$ 来求解递推关系式,代入上面得到:

$$a^2 - 2\gamma a + 1 = 0 \quad 或 \quad a = \gamma \pm \sqrt{\gamma^2 - 1} \qquad (5.5.12)$$

对所有解 $a$,当$|a|<1$时,递推是稳定的。此式成立(可验证)应有 $|\gamma|\leqslant 1$或 $n\leqslant x$。所以当 $n$ 很大时,不能用式(5.5.2)从 $J_0(x)$和 $J_1(x)$起始来计算 $J_n(x)$。

在这个问题上,若能有某些实际的定理作保证,就可能会更好些(虽然我们一直采取一中试探性的方法)。下面两个定理是由 Perron[2]给出的:

定理 $A$ 　式(5.5.9)中若 $a_n\sim an^\alpha$,$b_n\sim bn^\beta$(当 $n\to\infty$且 $\beta<2\alpha$ 时),则

$$g_{n+1}/g_n \sim - an^\alpha, \qquad f_{n+1}/f_n \sim - (b/a)n^{\beta-\alpha} \qquad (5.5.13)$$

$f_n$ 就是式(5.5.9)的最小解。

定理 $B$ 　与定理 $A$ 的条件相同,只是 $\beta=2\alpha$,考察下面**特征多项式**

$$t^2 + at - b = 0 \qquad (5.5.14)$$

若式(5.5.14)的根 $t_1$,$t_2$数值不等,不妨设 $|t_1|>|t_2|$则

$$g_{n+1}/g_n \sim t_1 n^\alpha, \quad f_{n+1}/f_n \sim t_2 n^\alpha \qquad (5.5.15)$$

$f_n$ 也是式(5.5.9)的最小解。若不具备上述两定理的条件,则不存在最小解。(关于递推稳定性更详细的讨论,参见文献[3])

若求得的解就是最小解,那么下面该接着怎办呢?正如一句古老的谚语所说的,事情总会是有好有坏:在某一方向上递推关系严重不稳定,则在其反方向上,(不希望的)解总是收

敛非常快。这意味着可以用连续项 $f_n$ 和 $f_{n+1}$ 的任意种子值开始，(在稳定方向上递推足够多步后)收敛到所求的函数与一个未知的规一化因子之积。若还存在其他对序列进行规一化的方法(如通过 $f_n$ 的求和公式)，则它确实是求函数值的一种实用的方法。这方法称为**米勒**(*Miller*)**算法**。通常式(5.5.2)的应用示例就是用这种方法，其规一化公式为

$$1 = J_0(x) + 2J_2(x) + 2J_4(x) + 2J_6(x) + \cdots \tag{5.5.16}$$

正好三项递推式和**连分式**之间有一个重要的关系。(5.5.9)递推关系可写成：

$$y_n/y_{n-1} = -b_n/(a_n + y_{n+1}/y_n) \tag{5.5.17}$$

从 $n$ 开始对上式反复迭代，得到：

$$\frac{y_n}{y_{n-1}} = -\frac{b_n}{a_n -}\,\frac{b_{n+1}}{a_{n+1} -}\cdots \tag{5.5.18}$$

由 Pincherle 定理[2]当且仅当式(5.5.9)存在最小解 $f_n$ 时，则式(5.5.18)收敛，此时收敛到 $f_n/f_{n-1}$。通常对于 $n=1$时的情况，还需结合一些其它方法求解 $f_0$，这些结果都是下一章计算各种特殊函数值的各种实用方法的基础。

### 5.5.2 克伦肖的递推公式

**克伦肖**(Clenshaw)**的递推公式**是计算求和式的一种优秀而有效的方法，该和式是系数乘上遵循递推公式的函数之积，例如，

$$f(\theta) = \sum_{k=0}^{N} c_k \cos(k\theta) \quad 或 \quad f(x) = \sum_{k=0}^{N} c_k P_k(x)$$

其原理如下：假设求和为

$$f(x) = \sum_{k=0}^{N} c_k F_k(x) \tag{5.5.19}$$

$F_k$ 满足递推关系

$$F_{n+1}(x) = \alpha(n,x)F_n(x) + \beta(n,x)F_{n-1}(x) \tag{5.5.20}$$

$\alpha(n,x)$ 和 $\beta(n,x)$ 为函数。现由以下递推式定义 $y_k$ $(k=N,N-1,\ldots,1)$：

$$\begin{aligned} &y_{N+2} = y_{N+1} = 0 \\ &y_k = \alpha(k,x)y_{k+1} + \beta(k+1,x)y_{k+2} + c_k \quad (k = N, N-1, \ldots, 1) \end{aligned} \tag{5.5.21}$$

为求 $c_k$ 解方程式(5.5.21)，然后整理和式(5.5.19)，得到如下形式(一部分)：

$$\begin{aligned} f(x) = &\cdots \\ &+ [y_8 - \alpha(8,x)y_9 - \beta(9,x)y_{10}]F_8(x) \\ &+ [y_7 - \alpha(7,x)y_8 - \beta(8,x)y_9]F_7(x) \\ &+ [y_6 - \alpha(6,x)y_7 - \beta(7,x)y_8]F_6(x) \\ &+ [y_5 - \alpha(5,x)y_6 - \beta(6,x)y_7]F_5(x) \\ &+ \cdots \\ &+ [y_2 - \alpha(2,x)y_3 - \beta(3,x)y_4]F_2(x) \\ &+ [y_1 - \alpha(1,x)y_2 - \beta(2,x)y_3]F_1(x) \\ &+ [c_0 + \beta(1,x)y_2 - \beta(1,x)y_2]F_0(x) \end{aligned} \tag{5.5.22}$$

注意，最后一行增加又减去一项 $\beta(1,x)y_2$。认真考察式(5.5.22)中含有 $y_k$ 因子的项，可以发现按式(5.5.20)的递推关系式其中和为零，其它 $y_k$ 类似，直到 $y_2$。最后式(5.5.22)所剩项为：

$$f(x) = \beta(1,x)F_0(x)y_2 + F_1(x)y_1 + F_0(x)c_0 \qquad (5.5.23)$$

式(5.5.21)和(5.5.23)是计算式(5.5.19)的 **clenshaw 递推公式**。先用式(5.5.21)递推计算 $y_k$，求得 $y_2$ 和 $y_1$ 后，用式(5.5.23)求得所求之和。

clenshaw 递推方法是按 $k$ 的递减顺序合并系数 $c_k$。每一步，所有以前 $c_k$ 的效果作为两个系数"记忆"下来，然后用它们乘函数 $F_{k+1}$ 和 $F_k$(最后为 $F_0$ 和 $F_1$)。当 $k$ 很大而函数 $F_k$ 很小，并且 $k$ 很小而 $c_k$ 也很小时，和式主要由小的 $F_k$ 决定。此时，被记忆的系数将被消去而失去重要性。例如求一个级数和

$$J_{15}(1) = 0 \times J_0(1) + 0 \times J_1(1) + \cdots + 0 \times J_{14}(1) + 1 \times J_{15}(1) \qquad (5.5.24)$$

其中 $J_{15}$ 很小，最终表示为消去 $J_0$ 和 $J_1$ 的线性组合，其 $J_0$ 和 $J_1$ 都是单位阶的。

这种情况下，解决办法是用另一种(Clenshaw)递推方式。即按上升顺序合并 $c_k$。相应的方程是：

$$y_{-2} = y_{-1} = 0 \qquad (5.5.25)$$

$$y_k = \frac{1}{\beta(k+1,x)}[y_{k-2} - \alpha(k,x)y_{k-1} - c_k]$$

$$(k = 0,1,\ldots,N-1) \qquad (5.5.26)$$

$$f(x) = c_N F_N(x) - \beta(N,x)F_{N-1}(x)y_{N-1} - F_N(x)y_{N-2} \qquad (5.5.27)$$

自动验证式(5.5.23)的第一个求和运算项，是否符号相反，并且数值是否基本相等，从而判断是否应该采用式(5.5.23)至(5.5.27)，而不能采用式(5.5.21)和(5.5.23)，但这种情况比较少见。除了这种特殊情况外，Clenshaw 递推总是稳定的，并且与函数 $F_k$ 的递推是在上升方向还是下降方向上的稳定性无关。

参考文献和进一步读物：


Abramowitz, M., and Stegun, I. A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, vol. 55, pp. xiii, 697. [1]

Gautschi, W. 1967, *SIAM Review*, vol. 9, pp. 24~82. [2]

Lakshmikantham, V., and Trigiante, D. 1988, *Theory of Difference Equations: Numerical Methods and Applications* (San Diego: Academic Press). [3]

Acton, F. S. 1970, *Numerical Methods That Work*; 1990, corrected edition, pp. 20ff. [4]

Clenshaw, C. W. 1962, *Mathematical Tables*, vol. 5, National Physical Laboratory (London: H. M. Stationery Office). [5]


## 5.6 二次方程和三次方程

简单代数方程的根可视为该方程系数的函数，这些函数我们在初等代数中学过，但奇怪的是，很多人不知道如何求解有两个实根的二次方程，或者如何求一个三次方程的实数解。

二次方程

$$ax^2 + bx + c = 0 \qquad (5.6.1)$$

$a$、$b$、$c$ 为实数，它的解有两种写法，即

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{5.6.2}$$

和

$$x = \frac{2c}{-b \pm \sqrt{b^2 - 4ac}} \tag{5.6.3}$$

如果用式(5.6.2)或(5.6.3)求根，则将自找麻烦：若 $a$ 或(和)$c$ 很小，则其中一根是用 $b$ 去减一个非常接近的量(即判别式)；这样得到的结果很不准确。正确的求解方式是计算

$$q \equiv -\frac{1}{2}\left[b + \mathrm{sgn}(b)\sqrt{b^2 - 4ac}\right] \tag{5.6.4}$$

则两根为

$$x_1 = \frac{q}{a} \quad \text{及} \quad x_2 = \frac{c}{q} \tag{5.6.5}$$

若系数 $a$、$b$、$c$ 为复数而不是实数，上面公式仍然有效，只是式(5.6.4)中平方根的符号必须使：

$$\mathrm{Re}(b^* \sqrt{b^2 - 4ac}) \geqslant 0 \tag{5.6.6}$$

其中 Re 表示实部，星号表示共轭复数。

关于二次方程，首先回忆一下反双曲函数，实际上它就是此方程解的对数

$$\sinh^{-1}(x) = \ln(x + \sqrt{x^2 + 1}) \tag{5.6.7}$$

$$\cosh^{-1}(x) = \pm \ln(x + \sqrt{x^2 - 1}) \tag{5.6.8}$$

$x \geqslant 0$时，式(5.6.7)是稳健的，当 $x$ 为负数时，利用对称性 $\sinh^{-1}(-x) = -\sinh^{-1}(x)$。而显然式(5.6.8)仅对 $x \geqslant 1$时才有效。

三次方程

$$x^3 + ax^2 + bx + c = 0 \tag{5.6.9}$$

$a$、$b$、$c$ 为实系数或复系数，首先计算

$$Q \equiv \frac{a^2 - 3b}{9} \quad \text{和} \quad R \equiv \frac{2a^3 - 9ab + 27c}{54} \tag{5.6.10}$$

若满足 $Q$、$R$ 是实数(当 $a$、$b$、$c$ 为实数时通常满足)和 $R^2 < Q^3$，则三次方程有三个实根。求根时先计算

$$\theta = \arccos(R / \sqrt{Q^3}) \tag{5.6.11}$$

由此得三个根为：

$$\begin{aligned} x_1 &= -2\sqrt{Q}\cos\left(\frac{\theta}{3}\right) - \frac{a}{3} \\ x_2 &= -2\sqrt{Q}\cos\left(\frac{\theta + 2\pi}{3}\right) - \frac{a}{3} \\ x_3 &= -2\sqrt{Q}\cos\left(\frac{\theta - 2\pi}{3}\right) - \frac{a}{3} \end{aligned} \tag{5.6.12}$$

这些方程式是在1615年出版的，Fransois Viete 的著作“*De emendatione*”第Ⅵ章中第一次出现。否则，计算

$$A = -\left[R + \sqrt{R^2 - Q^3}\right]^{1/3} \tag{5.6.13}$$

其中平方根的符号应满足

$$\mathrm{Re}(R^* \sqrt{R^2 - Q^3}) \geqslant 0 \tag{5.6.14}$$

(星号表示共轭复数。)若 Q 和 R 均为实数,式(5.6.13)~(5.6.14)等价于:

$$A = -\mathrm{sqn}(R)\left[|R| + \sqrt{R^2 - Q^3}\right]^{1/3} \tag{5.6.15}$$

其中假设平方根为正。下一步计算

$$B = \begin{cases} Q/A & (A \neq 0) \\ 0 & (A = 0) \end{cases} \tag{5.6.16}$$

由此可知三个根为:

$$x_1 = (A + B) - \frac{a}{3} \tag{5.6.17}$$

($a$,$b$,$c$ 为实数时唯一的实根)

$$\begin{aligned} x_2 &= -\frac{1}{2}(A + B) - \frac{a}{3} + i\,\frac{\sqrt{3}}{2}(A - B) \\ x_3 &= -\frac{1}{2}(A + B) - \frac{a}{3} - i\,\frac{\sqrt{3}}{2}(A - B) \end{aligned} \tag{5.6.18}$$

(同样为一对共轭复数。)式(5.6.13)~(5.6.16)的排列形式可将舍入误差降至最小,而且正如 A.J.Glassman 所指出的,还可以保证所选择计算立方根的方法,不会导致根的丢失。若要解一组仅系数稍有不同的三次方程组时,则用牛顿法(第9.4节)更有效。

参考文献和进一步读物:


Pachner, J. 1983, *Handbook of Numerical Analysis Applications* (New York: McGraw-Hill), §6.1.

Mckelvey, J.p. 1984, *American Journal of Physics*, vol. 52, pp. 269~270; see also vol. 53, p. 775, and vol. 55, pp. 374~375.


## 5.7 数值求导

若已求得一函数 $f(x)$,现在欲求其导数 $f'(x)$。很容易是不是?根据导数定义,$h \to 0$时求极限

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \tag{5.7.1}$$

上式表明:选一个很小的 $h$,计算 $f(x+h)$。$f(x)$值很可能已经有了,若还没有,也求一下。最后运用式(5.7.1)就可计算出导数。还有什么东西值得讨论的呢?

事实上还有很多方法。直接按上面过程求导数,几乎肯定得到的是不精确的结果。只有当函数 $f$ 值的计算极其费事,而又已经计算了 $f(x)$,并且只要最多再求一次函数值就可求得导数时,上述方法才可考虑使用。在这种情况下,剩下的任务就是寻找合适的 $h$。下面开始讨论:

式(5.7.1)式有两个误差源,截尾误差和舍入误差。截尾误差来源于泰勒级数扩展式的高次项:

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}h^2 f''(x) + \frac{1}{6}h^3 f'''(x) + \cdots \tag{5.7.2}$$

即有

$$\frac{f(x+h)-f(x)}{h} = f' + \frac{1}{2}hf'' + \ldots \tag{5.7.3}$$

而舍入误差来源较多。首先 $h$ 中有舍入误差；假定在 $x=10.3$ 处随便选定 $h=0.0001$，$x=10.3$ 及 $x+h=10.30001$ 都不是具有能用精确二进制表示的数。因此，都具有机器浮点格式 $\epsilon_m$（单精度时可达 $10^{-7}$）的某些相对误差。$h$ 的**有效值**误差，也就是机器表示的 $x+h$ 和 $x$ 之间的差值，其数量级为 $\epsilon_m x$，其中隐含量级为 $\epsilon_m x/h \sim 10^{-2}$ 的 $h$ 的相对误差。由式(5.7.1)即看出，导数中也存在同样大小的相对误差。

结论1：选择 $h$，使得 $x+h$ 和 $x$ 能被一个可表示的数精确地区分。通常用下面的程序步骤实现：

$$\begin{aligned} \text{temp} &= x + h \\ h &= \text{temp} - x \end{aligned} \tag{5.7.4}$$

某些优化的编译器和一些带浮点处理芯片的机器其内部精度高，这种方法就失效。此时，通常在式(5.7.4)的两式之间调用一次空函数 donothing(temp)，强制 temp 位于并超出可寻址存储器范围。

当 $h$ 为精确数时，式(5.7.1)舍入误差为 $e_r \sim \epsilon_f|f(x)/h|$，其中 $\epsilon_f$ 为计算 $f$ 的部分精度。对于简单的函数 $\epsilon_f$ 约等于机器精度，$\epsilon_f \approx \epsilon_m$；但对于计算复杂而且具有不至一个误差源的情况，误差可能比较大。式(5.7.3)的截尾误差数量级为 $e_t \sim |hf''(x)|$。变化 $h$ 使得和值 $e_r + e_t$ 为最小，这时 $h$ 最优值，

$$h \sim \sqrt{\frac{\epsilon_f f}{f''}} \approx \sqrt{\epsilon_f}\, x_c \tag{5.7.5}$$

其中 $x_c = (f/f'')^{1/2}$ 为函数 $f$ 的曲率尺度，或称其为变化的特性尺度。若无其它提示信息，则通常假定 $x_c = x$（除了在 $x=0$ 附近，此时应使用它 $x$ 尺度的典型估计值。）

使用式(5.7.5)后，计算导数的相对误差为

$$(e_r + e_t)/|f'| \sim \sqrt{\epsilon_f}\,(ff''/f'^2)^{1/2} \sim \sqrt{\epsilon_f} \tag{5.7.6}$$

上式中最后等式假定 $f$、$f'$、$f''$ 都具有相同的特性长度尺度，通常也正是如此。可见看出，简单的有限差分方程式(5.7.1)，最多能精确到机器精度 $\epsilon_m$ 的平方根。

如果每次求导数时可以求两个函数值，运用下面的对称形式效果会好得多：

$$f'(x) \approx \frac{f(x+h)-f(x-h)}{2h} \tag{5.7.7}$$

此时由式(5.7.2)得，截尾误差为 $e_t \sim h^2 f'''$。舍入误差与前面一样，由类似关面的简短计算，最优的 $h$ 值为

$$h \sim \left(\frac{\epsilon_f f}{f'''}\right)^{1/3} \sim (\epsilon_f)^{1/3} x_c \tag{5.7.8}$$

相对误差为

$$(e_r + e_t)/|f'| \sim (\epsilon_f)^{2/3} f^{2/3} (f''')^{1/3}/f' \sim (\epsilon_f)^{2/3} \tag{5.7.9}$$

上式与式(5.7.6)相比，一般情况比较精确一个数量级（单精度）或2个数量级（双精度）。由此得出结论2：选择 $h$ 等于 $\epsilon_f$ 或 $\epsilon_m$ 的正确次幂乘以的特征尺度 $x_c$。

对其它情况很容易导出正确的幂次，例如对二维函数，混合导数公式为：

$$\frac{\partial^2 f}{\partial x \partial y} \approx \frac{[f(x-h,y)-f(x,y)]-[f(x,y-h)-f(x,y)]}{h^2} \quad (5.7.10)$$

正确尺度是 $h \sim \epsilon_f^{1/4} x_c$。

没有简单的有限差分公式,如式(5.7.1)或(5.7.7)能给出机器的精度 $\epsilon_m$ 的公式,或者甚至比求 $f$ 值的估算精度 $\epsilon_f$ 更差的精度公式,这一点很令人失望。有没有更好的方法呢?

有的,不过要涉及到考察相对于 $x_c$ 尺度上函数的行为,还得假定函数平滑,或可解析的,这样泰勒展开式(6.7.2)中的高次项才有意义。同时还涉及求函数值的多种方法,所以精度的提高是以计算的费用增大为代价的。

Richardson 的延迟极限逼近思想很有吸引力。数值积分中龙贝格积分就来自这种思想,(见第4.3节)。求导数时,可将由越来越小的 $h$ 值求得的结果外推至 $h \to 0$。由尼维尔算法(第3.1节),应用每次有限差分新的计算结果求得高阶的外推。**Ridders**[2]漂亮地实现了这种思想,下面的程序 **dfridr** 就是基于他的算法,只是修改了终止条件。程序的输入是函数 $f$(称 func),正数 $x$,最大步长 $h$(与 $h$ 相比它更象前面称为 $x_c$ 的量)。输出为导数的返回值及误差估计值 err。

```
#include <math.h>
#include "nrutil.h"
#define CON 1.4                    每次迭代步长减小 CON
#define CON2 (CON * CON)
#define BIG 1.0e30
#define NTAB 10                    设置表的最大尺度
#define SAFE 2.0                   当误差 SAFE 比目前求得最好结果还坏时返回

float dfridr(float (*func)(float), float x, float h, float *err)
  返回由 Ridders 多项式外推法求得的函数 func 在点 x 处的导数值.输入 h 为估计初始步长,不必很小,而必须为能
  够使 func 显著改变变量 x 的增量。导数的误差估计值的返回值为 err.
{
    int i,j;
    float errt, fac, hh, **a,ans;

    if (h == 0.0) nrerror("h must be nonzero in dfridr.");
    a=matrix(1,NTAB,1,NTAB);
    hh=h;
    a[1][1]=((*func)(x+hh)-(*func)(x-hh))/(2.0*hh);
    *err=BIG;
    for (i=2;i<=NTAB;i++) {              尼维尔表中的连续列将进入更小的步长和更高阶的外推
        hh /= CON;
        a[1][i]=((*func)(x+hh)-(*func)(x-hh))/(2.0*hh);         试验新的更小步长
        fac=CON2;
        for (j=2;j<=i;j++) {             不需再求函数值,计算不同阶的外推
            a[j][i]=(a[j-1][i]*fac-a[j-1][i-1])/(fac-1.0);
            fac=CON2*fac;
            errt=FMAX(fabs(a[j][i]-a[j-1][i]),fabs(a[j][i]-a[j-1][i-1]));
                  误差策略是将每次新的外推结果与低一阶结果比较,包括当前步长及前一次步长的比较
               if (errt <= *err) {         若误差减小,保存改进的结果
                  *err=errt;
                  ans=a[j][i];
               }
        }
        if (fabs(a[i][i]-a[i-1][i-1]) >= SAFE*(*err)) break;     若高阶结果比 SAFE 差……
    }
    free_matrix(a,1,NTAB,1,NTAB);
    return ans;
}
```

程序 **dfridr** 中求函数 func 值的次数一般是6到12次，但可大到2 * NTAB 次，h 为输入量，h 越大求得精度也较好，直至在某突变点处，无意义的外推导致一个很大的误差返回值。所以必须选择适当大小的 h，注意返回值 err，若这个返回值不够小时减小 h。函数的特征 x 尺度为单位数量级时，一般典型地选 h 为十分之几。

除 Ridders 方法外还有其它一些方法。若函数很平滑，并且在某区间内任意点处要多次计算函数的导数，此时最好在此区间内构造函数的切比雪夫多项式逼近式，直接由求得的切比雪夫系数求导数。下面第5.8~5.9节将讨论这种方法。

另外还有一种方法，适用于在区间的等分点上的数据表组成(可能有噪声)的函数。在每个所求点 $x$ 值处，用表中该点左边 $n_L$ 个点及右边 $n_R$ 个点，由最小二乘方拟合 $M$ 阶多项式。导数估计值即为求得的拟合多项式的导数。Savitzky-Golay 平滑滤波器用于构造多项式时很有效，第14.8节中再讨论。那里将给出一个程序求滤波器系数，这些系数不仅可以构造拟合多项式，而且还可用来计算数据点乘滤波器系数和的累积值。给定的程序 **savgol** 中有一个参数为 ld，它表示求拟合多项式的哪一阶导数。求第一阶导数时置 ld＝1，导数值是累积和除以采样区间 h。

参考文献和进一步读物：


Dennis, J. E., and Schnabel, R. B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* (Englewood Cliffs, NJ: Prentice-Hall), §§5.4~5.6. [1]

Ridders, C. J. F. 1982, *Advances in Engineering Software*, vol. 4, no. 2, pp. 75~76. [2]


## 5.8 切比雪夫逼近

$n$ 阶切比雪夫多项式用 $T_n(x)$ 表示，写成公式为

$$T_n(x) = \cos(n \arccos x) \tag{5.8.1}$$

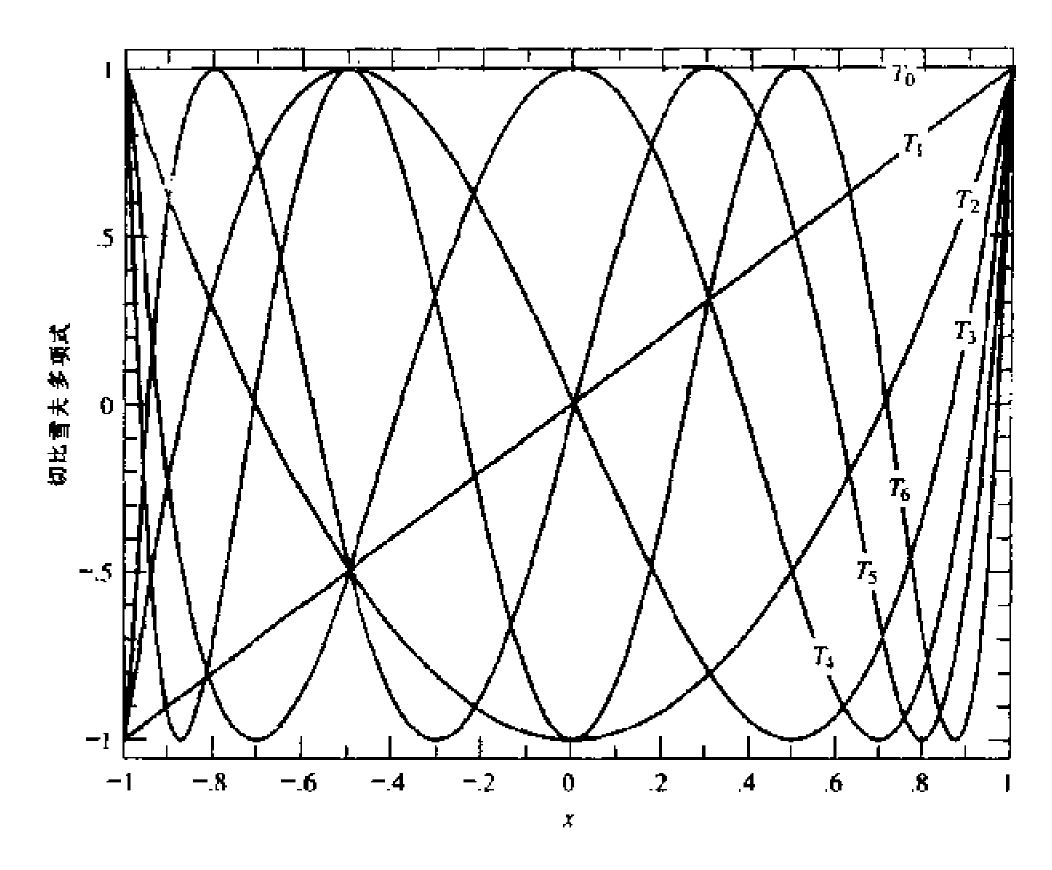
看上去它像三角函数公式(事实上，切比雪夫多项式和离散傅里叶变换有紧密联系)，结合三角恒等式由式(5.8.1)可得 $T_n(x)$ 的显式为(见图5.8.1)：

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_2(x) &= 2x^2 - 1 \\ T_3(x) &= 4x^3 - 3x \\ T_4(x) &= 8x^4 - 8x^2 + 1 \\ &\cdots \\ T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x) \qquad n \geqslant 1 \end{aligned} \tag{5.8.2}$$

(也存在上式反变换公式，即用 $T_n(x)$ 项表示的 $x$ 幂次公式，见公式(5.11.2)~(5.11.3))。

切比雪夫多项式在[－1,1]区间上对权 $(1-x^2)^{-1/2}$ 是正交的，特别是

$$\int_{-1}^{1} \frac{T_i(x)T_j(x)}{\sqrt{1-x^2}} dx = \begin{cases} 0 & i \neq j \\ \pi/2 & i = j \neq 0 \\ \pi & i = j \end{cases} \tag{5.8.3}$$




注意：$T_j$ 在[−1,1]区间上有 $j$ 个根，且多项式值介于±1之间。

图5.8.1 切比雪夫多项式 $T_0(x)$到 $T_6(x)$。

多项式 $T_n(x)$在[−1,1]区间上有 $n$ 个零点，分布在如下的点上

$$x = \cos\left(\frac{\pi(k-\frac{1}{2})}{n}\right) \qquad k = 1,2,\ldots,n \tag{5.8.4}$$

在同样的区间上，有 $n+1$ 个极值(极大值和极小值)，其分布在如下的点上

$$x = \cos\left(\frac{\pi k}{n}\right) \qquad k = 0,1,\ldots,n \tag{5.8.5}$$

所有极大值处 $T_n(x)=1$，所有极小处 $T_n(x)=-1$，正是这种性质使切比雪夫多项式在函数多项式逼近中非常有用。

切比雪夫多项式不仅满足连续正交关系式(5.8.3)，而且满足离散正交关系：若 $x_k(k=1,\cdots,m)$为由式(5.8.4)给出的 $T_m(x)$的 $m$ 个零点，且 $i,j<m$，则

$$\sum_{k=1}^{m} T_i(x_k)T_j(x_k) = \begin{cases} 0 & i \neq j \\ m/2 & i = j \neq 0 \\ m & i = j = 0 \end{cases} \tag{5.8.6}$$

综合式(5.8.1)，(5.8.4)和(5.8.6)，不难证明下面的定理：设 $f(x)$为[−1,1]区间上任意函数，定义 $N$ 个系数 $c_j$，$j=0,\cdots,N-1$为：

$$c_j = \frac{2}{N}\sum_{k=1}^{N} f(x_k)T_j(x_k)$$

$$= \frac{2}{N}\sum_{k=1}^{N} f\left[\cos\left(\frac{\pi(k-\frac{1}{2})}{N}\right)\right]\cos\left(\frac{\pi j(k-\frac{1}{2})}{N}\right) \tag{5.8.7}$$

那么逼近公式为

$$f(x) \approx \left[\sum_{k=0}^{N-1} c_k T_k(x)\right] - \frac{1}{2}c_0 \tag{5.8.8}$$

在 $T_N(x)$ 的 $N$ 个零点处，上式精确相等。

$N$ 确定时，式(5.8.8)就是[−1,1]区间内逼近 $f(x)$ 的多项式($T_N(x)$ 的所有零点也在此区间内)。为什么说这种特殊多项式逼近比在此区间内其它任何严格的逼近函数要好呢？这并不是因为式(5.8.8)比其它 $n$ 阶多项式更精确(对特定意义上的精确性来说)，而是因为式(5.8.8)可非常漂亮地截为低阶($m \ll N$)多项式，能得到 $m$ 阶最好的逼近(在某种准确性的定义意义上来说)。设 $N$ 很大，以致可认为式(5.8.8)是 $f(x)$ 的完全逼近。现考察截尾逼近

$$f(x) \approx \left[\sum_{k=0}^{m-1} c_k T_k(x)\right] - \frac{1}{2}c_0 \tag{5.8.9}$$

$c_j$ 同样由式(5.8.7)得到。因 $T_k(x)$ 限定在 ±1 之间，故式(5.8.9)与(5.8.8)之差不可能大于舍弃的 $c_k$($k=m,\cdots,N-1$)之和。若 $c_k$ 衰减很快(通常如此)，则误差主要由 $c_m T_m(x)$ 确定，$c_m T_m(x)$ 为一振荡函数，在区间[−1,1]上平滑地分布着 $m+1$ 个相同的极值。这种误差的平滑扩散性质很重要，这样式(5.8.9)的切比雪夫多项式就是我们想得到的逼近多项式——极小极大多项式，它在所有相同阶数的多项式中，与原函数 $f(x)$ 的最大偏差达到最小。极小极大多项式很难找；而切比雪夫逼近多项式几乎是恒等而且很容易计算。

给定计算函数 $f(x)$ 值的一些方法后，须找到算法以实现式(5.8.7)，和(检验得到的 $c_k$ 和截尾值 $m$ 后)求式(5.8.9)之值。以后用此式求 $f(x)$ 的值就很方便了。

第一项任务可直接计算。这里用式(5.8.7)的一般推广形式，即允许在任意两个数 $a$ 和 $b$ 范围内逼近，而不仅仅限于−1到1之间。对此可按下面进行变量变换

$$y \equiv \frac{x - \frac{1}{2}(b+a)}{\frac{1}{2}(b-a)} \tag{5.8.10}$$

以及实现对 $y$ 的 $f(x)$ 的切比雪夫多项式逼近。

```
#include <math.h>
#include "nrutil.h"
#define PI 3.141592653589793

void chebft(float a, float b, float c[], int n, float (*func)(float))
    切比雪夫拟合:给定函数 func、区间上下极限[a,b]、最大阶数 n,本程序先计算 n 个系数 c[0..n-1],使 func(x) ≈
    [∑_{k=0}^{n-1} c_k T_k(y)]-c_0/2,其中 y 与 x 关系由式(5.8.10)确定。调用时 n 大小选取应适当(如30到50),然后在较小
    的 m 值处截去 c 数组,使 c_m 及其以后元素均可忽略
{
    int k,j;
    float fac,bpa,bma,*f;
```

```
    f=vector(0,n-1);
    bma=0.5*(b-a);
    bpa=0.5*(b+a);
    for (k=0;k<n;k++) {                     由式(5.8.7)求函数在n个点上的值
        float y=cos(PI*(k+0.5)/n);
        f[k]=(*func)(y*bma+bpa);
    }
    fac=2.0/n;
    for (j=0;j<n;j++) {
        double sum=0.0;                     按双精度累积式，一般双精度可忽略
        for (k=0;k<n;k++)
            sum += f[k]*cos(PI*j*(k+0.5)/n);
        c[j]=fac*sum;
    }
    free_vector(f,0,n-1);
}
```

若发现程序 **chebft** 执行大部分时间是在计算 $N^2$ 次余弦，而不是求 $N$ 次函数值，此时应参见第12.3节，尤其是式(12.3.22)，讨论如何用快速余弦变换方法求式(5.8.7)值。

有了切比雪夫系数值后，又如何求逼近值呢？有人可能会用式(5.8.2)递推关系从 $T_0=1$ 和 $T_1=x$ 来求 $T_k(x)$，再求式(5.8.9)之和。更好的办法是用切比雪夫递推公式(第5.4节)，同时完成两个过程。应用于切比雪夫级数(5.8.9)的递推为

$$
\begin{aligned}
d_{m+1} &\equiv d_m \equiv 0 \\
d_j &= 2xd_{j+1} - d_{j+2} + c_j \qquad j = m-1, m-2, \ldots, 1 \\
f(x) \equiv d_0 &= xd_1 - d_2 + \frac{1}{2}c_0
\end{aligned}
\tag{5.8.11}
$$

```
float chebev(float a, float b, float c[], int m, float x)
    切比雪夫求值：参数均为输入，c[0..m-1]为切比雪夫系数数组c中前m个元素，应由chebft(调用时a,b应相同)
    输出。在y=[x-(b+a)/2]/[(b-a)/2]处求值切比雪夫多项式[∑(k=0..m-1) c_k T_k(y)]-c_0/2，返回的结果是函数值
{
    void nrerror(char error_text[]);
    float d=0.0,dd=0.0,sv,y,y2;
    int j;

    if ((x-a)*(x-b) > 0.0) nrerror("x not in range in routine chebev");
    y2=2.0*(y=(2.0*x-a-b)/(b-a));                     变量替换
    for (j=m-1;j>=1;j--) {                            切比雪夫递推式
        sv=d;
        d=y2*d-dd+c[j];
        dd=sv;
    }
    return y*d-dd+0.5*c[0];                           最后一步是不同的
}
```

若区间[−1,1]上的逼近函数为偶函数，其展开式只包含偶次切比雪夫多项式。奇系数全为零时，调用程序 **chebev** 是没用的[1]。对式(5.8.1)应用半角恒等式，得到关系式

$$T_{2n}(x) = T_n(2x^2-1) \tag{5.8.12}$$

这样，可将偶系数连续存放在数组 c 中，而变量 $x$ 由 $2x^2-1$ 代替，然后调用 **chebev** 函数后，就可求得一系列偶切比雪夫多项式的值。

奇函数的展开式则只有奇切比雪夫多项式。最好是将其改写为函数 $f(x)/x$ 的扩展式，这样仅含偶切比雪夫多项式。在 $x=0$ 附近可求得精确值。$f(x)/x$ 的系数 $c_n'$ 可由 $f(x)$ 的系

数按下面递推式求得：

$$c'_{N+1}=0$$
$$c'_{n-1}=2c_n-c'_{n+1} \qquad n=N,N-2,\ldots \qquad (5.8.13)$$

式(5.8.13)服从式(5.8.2)的递推关系式。

若一定要求奇切比雪夫级数的值，较好的方法是再用 $y=2x^2-1$ 替换 $x$，并将奇系数逐续存放在数组 c 中。现将(5.8.11)中最后一个公式改为

$$f(x)=x[(2y-1)d_2-d_3+c_1] \qquad (5.8.14)$$

并修改程序 **chebev** 中相应的行。

参考文献和进一步读物：


Clenshaw, C. W. 1962, *Mathematical Tables*, vol. 5, National Physical Laboratory, (London; H. M. Stationery Office). [1]


## 5.9 切比雪夫逼近函数的微分和积分

如果得到某一范围内逼近一个函数的切比雪夫系数之后(如由第5.8节的 **chebft**)，把它们变换成与该函数微分和积分相对应的切比雪夫系数就是件简单的事了。完成这种变换后，就可以计算其导数与积分值，犹如一开始就在做拟合切比雪夫函数似的。

相应的公式为：设 $c_i, i=0,\cdots,m-1$ 为式(5.8.9)中逼近函数 $f$ 的系数，$C_i$ 为逼近 $f$ 的不定积分的系数，$c'_i$ 为逼近 $f$ 的导数的系数，则有

$$C_i=\frac{c_{i-1}-c_{i+1}}{2i} \qquad (i>1) \qquad (5.9.1)$$

$$c'_{i-1}=c'_{i+1}+2(i-1)c_i \qquad (i=m-1,m-2,\ldots,2) \qquad (5.9.2)$$

相应于积分常数的任意性，扩展式(5.9.1)中 $C_0$ 选择也是任意的。递推式(5.9.2)起始项 $c'_m=c'_{m+1}=0$，这些值表示关于原函数的第 $m+1$ 项切比雪夫多项式系数没有信息。

下面是实现式(5.9.1)与(5.9.2)的子程序

```
void chder(float a, float b, float c[], float cder[], int n)
    a,b,c[0..n-1]为第5.8节中 chebft 子程序的输出，并n给定，它是设计的逼近阶数(即c的长度)。本程序返回数组
    cder[0..n-1]，是系数为c的函数导数的切比雪夫系数。
{
    int j;
    float con;

    cder[n-1]=0.0;                                   n-1和n-2是特殊情况
    cder[n-2]=2*(n-1)*c[n-1]
    for (j=n-3;j>=0;j--)
        cder[j]=cder[j+2]+2*(j+1)*c[j+1];            方程式(5.9.2)
    con=2.0/(b-a);
    for (j=0;j<n;j++)                                区间b-a内规一化
        cedr[j] *= con
}
```

```
void chint(float a, float b, float c[], float cint[], int n)
    给定a,b,c[0..n-1]为第5.8节程序 chebft 的输出，并给定n为要求逼近阶数(c使用的长度)，本程序返回系数为
    c的函数积分的切比雪夫系数数组cint[0..n-1]，选择积分常数使得积分在a处为零。
```

```
{
    int j;
    float sum=0.0,fac=1.0,con;

    con=0.25*(b-a);                                   区间 b-a 内的规一化因子
    for (j=1;j<=n-2;j++) {
        cint[j]=con*(c[j-1]-c[j+1])/j;                式(5.9.1)

        sum += fac*cint[j];                           累积积分常数
        fac = -fac;                                   将等于±1
    }
    cint[n-1]=con*c[n-2]/(n-1);                       对于 j=n-1,(5.9.1)式的特殊情况
    sum += fac*cint[n-1];
    cint[0]=2.0*sum;                                  设置积分常数
}
```

### 5.9.1 Clenshaw-Curtis 积分

因为平滑函数的切比雪夫系数 $c_i$ 一般按指数迅速减小，故(5.9.1)式通常是一种高效求积分方法的基础。若需在 $a \leqslant x \leqslant b$ 范围中，对 $x$ 的许多不同值求函数的积分值，则可按顺序使用程序 **chebft** 和 **chint**，接着反复调用 **chebev**。

若仅求单个定积分 $\int_a^b f(x)dx$，由(5.9.1)式可导出，此时 **chint** 和 **chebev** 需由下面简单公式替换：

$$\int_a^b f(x)dx = (b-a)\left[\frac{1}{2}c_1 - \frac{1}{3}c_3 - \frac{1}{15}c_5 - \cdots - \frac{1}{(2k+1)(2k-1)}c_{2k+1} - \cdots\right] \quad (5.9.3)$$

其中的 $c_i$ 由程序 **chebft** 求出。当 $c_{2k+1}$ 可忽略时，截断级数，并且第一个忽略项为估计误差。

这种方法称为 **Clenshaw-Curtis 积分方法**[1]。通常还能自适应选择 $N$，它是式(5.8.7)计算切比雪夫系数的个数，同时也是函数 $f(x)$ 的求值次数。若 $N$ 的选择使得式(5.9.3)中 $c_{2k+1}$ 不是足够小，则应试一下 $N$ 大一点的值。此时更好的方法是，用称为"梯形"或 Gauss-Lobatto(第4.5节)变量来替换式(5.8.7)，

$$c_j = \frac{2}{N}\sum_{k=0}^{N}{}'' f\left[\cos\left(\frac{\pi k}{N}\right)\right]\cos\left(\frac{\pi(j-1)k}{N}\right) \qquad j=1,\ldots,N \quad (5.9.4)$$

上式中的两撇表示和式中的第一项和最后一项都要乘以1/2。若式(5.9.4)中 $N$ 值增加1倍，则求函数值的点有一半与前一次一样，这样前面计算的函数值可重新使用。这种特性，再加上权和坐标的解析性(式(5.9.4)中的余弦函数)，使得 Clenshaw-Curtis 求积方法比高阶自适应高斯求积方法(见第4.5节)稍好一些。

若问题中要求 $N$ 值很大，此时应注意到，同时对所有 $j$ 值，通过快速余弦变换可以很快求得式(5.9.4)之值。(见第12.3节，尤其是式(12.3.17))。(已讨论过式(5.8.7)的非梯形形式也可以由快速余弦方法求出，参见式(12.3.22)。)

参考文献和进一步读物：


Clenshaw, C. W., and Curtis, A. R. 1960, *Numerische Mathematik*, vol. 2, pp. 197~205. [1]


## 5.10 切比雪夫系数的多项式逼近

读过前面两节，我们可能会问，"是否对变换后的变量 $y$ 必须存储和计算这个切比雪夫系数数组的切比雪夫逼近式。难道不能将 $c_k$ 转换成原变量 $x$ 的实际多项式的系数，而得到下面的多项式逼近形式呢？"

$$f(x) \approx \sum_{k=0}^{m-1} g_k x^k \quad (5.10.1)$$

是的，可以这样做(为此，我们还将提供算法)，但必须提醒注意：对(5.10.1)式的求值

(其中系数 $g$ 隐含了切比雪夫逼近)通常比直接对切比雪夫和式求值需要更多的有效数字(见程序 **chebev**)。这是由于切比雪夫多项式本身有一种很巧妙的消去性质:如 $T_n(x)$ 的首项系数为 $2^{n-1}$,$T_n(x)$ 的其它系数甚至更大;但经它们组合而成的多项式界于 ±1 之间。仅当 $m$ 不大于7或8时,才能考虑将切比雪夫逼近式写成直接多项式形式。即使这样,也还必须能容许精度比机器的舍入极限低2个左右的有效数字。

在式(5.10.1)中由 **Chebft**(在恰当的 $m$ 值处适当地截尾)输出的 $c$ 来求得 $g$,这只须顺序地调用下面两个函数:

```
#include "nrutil.h"

void chebpc(float c[], float d[], int n)
```

切比雪夫多项式系数。给定系数数组 c[0..n-1],本程序生成一个系数数组 d[0..n-1],使得 $\sum_{k=0}^{n-1} d_k y^k = \sum_{k=0}^{n-1} c_k T_k(y) - c_0/2$。方法是克伦肖递推式(5.8.11),只是代数地而不是算术地应用。

```
{
    int k,j;
    float sv,*dd;

    dd=vector(0,n-1);
    for (j=0;j<n;j++) d[j]=dd[j]=0.0;
    d[0]=c[n-1];
    for(j=n-2;j>=1;j--) {
        for (k=n-j;k>=1;k--) {
            sv=d[k];
            d[k]=2.0*d[k-1]-dd[k];
            dd[k]=sv;
        }
        sv=d[0];
        d[0] = -dd[0]+c[j];
        dd[0]=sv;
    }
    for (j=n-1;j>=1;j--)
        d[j]=d[j-1]-dd[j];
    d[0] = -dd[0]+0.5*c[0];
    free_vector(dd,0,n-1);
}


void pcshft(float a, float b, float d[], int n)
```

多项式系数转换。给定系数组 d[0..n-1],本程序生成系数数组 g[0..n-1],使得 $\sum_{k=0}^{n-1} d_k y^k = \sum_{k=0}^{n-1} g_k x^k$,其中 $x$,$y$ 的关系由(5.8.10)确定,即将 $-1<y<1$ 区间映射到 $a<x<b$ 区间上,数组 g 返回到 d 中。

```
{
    int k,j;
    float fac,cnst;

    cnst=2.0/(b-a);
    fac=cnst;
    for (j=1;j<n;j++) {           先用因子 cnst 调整比例
        d[j] *= fac;
        fac *= cnst;
    }
    cnst=0.5*(a+b);               然后重新定义所期望的转换
    for (j=0;j<=n-2;j++)          这里用综合除法完成转换。综合除法是中学代数课内容,
        for (k=n-2;k>=j;k--)      尚没有学过的读者,仍可以照此做下去,不用发愁
            d[k] -= cnst*d[k+1];
```

## 5.11 幂级数化简

切比雪夫方法的比较特殊的应用例子是**幂级数化简**，这是一种很有用的方法，看上去有点无中生有。

假定用收敛幂级数求函数值，例如

$$f(x) \equiv 1 - \frac{x}{3!} + \frac{x^2}{5!} - \frac{x^3}{7!} + \cdots \tag{5.11.1}$$

（这函数实际上就是 $\sin\sqrt{x}/\sqrt{x}$，不过我们还是假定不知道）。读者可能要求在某特定区间如$[0,(2\pi)^2]$内，多次求此级数值。那么，现在就要找到级数中某一项，使得在此项之前的有限级数引起的误差（近似等于忽略项的第一项）是允许的。在上面示例中，若 $x \leqslant (2\pi)^2$，第一个小于$10^{-7}$的项为 $x^{13}/(27!)$，因此此项近似等于最后一项为 $x^{12}/25!$的有限级数的误差。

注意，因为 $x^{13}$的指数很大，因此在特定区间中除了在 $x$ 很大的地方外，误差处处比$10^{-7}$小得多。此特性便于"简化"：若我们让区间内误差大到等于第一个忽略项在此区间端点处的值，那么就可以用一个很短的级数替代 13 项级数了。

分下面几步完成：

1. 将 x 变换到变量 $y$，如(5.8.10)式，将 $x$ 区间映射至 $-1 \leqslant y \leqslant 1$。
2. 求正好等于截尾级数（项数足够多可满足精度）的切比雪夫和式（如(5.8.8)式）的系数。
3. 将切比雪夫级数截成项数少的级数，用第一个被忽略的切比雪夫多项式的系数作为估计误差。
4. 转换成 $y$ 的多项式。
5. 变量再转换回 $x$。

当给定原级数展开式的系数，上面各步都可以数值计算。第一步正好是程序 **pcshft**（第5.10）的逆过程，那里是将 $y$ 的多项式（区间$[-1,1]$）映射为 $x$ 的多项式（区间$[a,b]$）。因为式(5.8.10)为 $x$ 和 $y$ 的线性关系式，所以也可将 **pcshft** 用于此逆过程。

$$\text{pcshft}(a,b,d,n)$$

的逆过程是（可以验证一下）：

$$\text{pcshft}\left(\frac{-2-b-a}{b-a}, \frac{2-b-a}{b-a}, d, n\right)$$

第二步是求程序 **chebpc**（将切比雪夫系数用于多项式系数）的逆操作。下面给出的程序 **pccheb** 正是完成此过程，应用公式[1]

$$x^k = \frac{1}{2^{k-1}}\left[T_k(x) + \binom{k}{1}T_{k-2}(x) + \binom{k}{2}T_{k-4}(x) + \cdots\right] \tag{5.11.2}$$

上式中最后一项依 $k$ 是奇数还是偶数而定，

$$\cdots + \binom{k}{(k-1)/2}T_1(x) \quad (k\text{ 偶数}), \qquad \cdots + \frac{1}{2}\binom{k}{k/2}T_0(x) \quad (k\text{ 奇数}) \tag{5.11.3}$$

```
void pccheb(float d[], float c[], int n)
```

程序 **chebpc** 的逆过程：给定多项式系数组 d[0..n-1]，返回等价切比雪夫系数数组 c[0..n-1]。

```
{
    int j,jm,jp,k;
    float fac,pow;

    pow=1.0;                                 2的幂次
```

```
c[0]=2.0*d[0];
for (k=1;k<n;k++) {                              在多项式的 x 阶次上循环
    c[k]=0.0;                                    寻找对应切比雪夫阶次
    fac=d[k]/pow;
    jm=k;
    jp=1;
    for (j=k;j>=0;j-=2,jm--,jp++) {              循环,以求具有组合系数
                                                 乘以 d[k]的切比雪夫的低阶项,见文中公式
        c[j] += fac;
        fac *= ((float)jm)/((float)jp);
    }
    pow += pow;
}
```

第四步和第五步分别由 **chebpc** 和 **pcshft** 完成,整个过程表述为下面形式:

```
#define NFEW ..
#define NMANY ..

    float *c, *d, *e,a,b;               在区间(a,b)中将系数为 e[0..NMANY-1]的幂级数

    c=vector(0,NMANY-1);                化简成 NFEW 个系数 d[0..NFEW-1]
    d=vector(0,NFEW-1);
    e=vector(0,NMANY-1);
    pcshft((-2.0-b-a)/(b-a),(2.0-b-a)/(b-a),e,NMANY);
    pccheb(e,c,NMANY);
    ...
                       这里一般需检查切比雪夫系数 c[0..NMANY-1]以决定 NFEM 该多小
    chebpc(c,d,NFEW);
    pcshft(a,b,d,NFEW);
```

上面例题中,第八至第十个切比雪夫系数算出数量级分别为 $-7\times10^{-6}$, $3\times10^{-7}$, $-9\times10^{-8}$,这样适当截尾后允许误差落在$10^{-6}$范围之内(单精度计算),得到的是 8~10项的多项式,而不是原来的 13 项。

13项的多项式可用一个 10 项多项式替换而没有减少精度。看上去确实有点无中生有。有什么魔法吗?没有。决定函数 $f(x)$是 13 项多项式。而与化简级数等价的是,我们可以在感兴趣的区间内,用第5.8中的方法,在足够多点上,构造切比雪夫逼近式来替换求函数 $f(x)$值。这样也能得到同样低阶多项式。主要是因为,切比雪夫系数的收敛速率与幂级数系级数的收敛速率无关,而只与事前[illegible]决定多项式逼近式所需要的项数。函数在某感兴趣的区域内可能有发散级数,但若函数本身平滑,则一定存在性能优良的多项式逼近式,可由第5.8节的方法找出,不要由级数化简法找。

参考文献和进一步读物:


Arfken, G. 1970, *Mathematical Methods for Physicists*, 2nd ed. (New York: Academic Press), p. 631. [1]


## 5.12 Padé 逼近

称为 Padé **逼近**是一种(特定阶的)有理函数,其幂级数展开式与给定级数的最高可能阶相同,若有理函数为

$$R(x) \equiv \frac{\sum_{k=0}^{M} a_k x^k}{1+\sum_{k=1}^{N} b_k x^k} \tag{5.12.1}$$

则称 $R(x)$ 为下面级数

$$f(x) = \sum_{k=0}^{\infty} c_k x^k \tag{5.12.2}$$

的 Padé 逼近式

若

$$R(0) = f(0) \tag{5.12.3}$$

且

$$\frac{d^k}{dx^k}R(x)|_{x=0} = \frac{d^k}{dx^k}f(x)|_{x=0}, \quad k = 1,2,\ldots,M+N \tag{5.12.4}$$

式(5.12.3)及(5.12.4)组成 $M+N+1$ 个方程可以求未知数 $a_0,\ldots,a_M$ 和 $b_0,\ldots,b_N$。得到这些方程最简单的方法是,将式(5.12.1)和式(5.12.2)等起来,两边同时乘以式(5.12.1)的分母,再对系数含 $c$、$b$ 和 $x$ 幂次项列成等式。我们考察 $M=N$ 的对角有理逼近的特殊情况(参见第3.2),则有 $a_0=c_0$,剩下 $a$、$b$ 满足

$$\sum_{m=1}^{N} b_m c_{N-m+k} = -c_{N+k}, \qquad k = 1,\ldots,N \tag{5.12.5}$$

$$\sum_{m=0}^{k} b_m c_{k-m} = a_k, \qquad k = 1,\ldots,N \tag{5.12.6}$$

(注意式(5.12.1)中 $b_0=1$)。解方程时从式(5.12.5)解起,因为它是一组对所有未知数 $b$ 的线性方程。尽管方程组为托普雷兹矩阵矩阵形式(参见式(2.8.8)),但经验表明方程通常表现为退化形式,因此不要用第2.8节方法求解,而用 $LU$ 分解法。此外,用迭代改进修正解是一个好主意(第2.5节中程序 **mprove**)。

一旦 $b$ 已知,则由式(5.12.6)得到求解未知数 $a$ 的明晰公式,就可完成求解。

Padé 逼近式典型地用于求某未知基础函数 $f(x)$。假定要计算 $f(x)$ 及 $f(x)$ 在 $x=0$ 的导数值:$f(0)$、$f'(0)$、$f''(0)$ 等等,而且可能必须由繁琐的解析展开式计算得到。当然,这些只是 $f(x)$ 幂级数展开式的前面几项;这些项数不必要变少,因为也不知道幂级数在哪里(是否会)收敛。

与切比雪夫逼近式(第5.8节),或者幂级数化简(第5.11节),这类仅已知的函数信息的方法相比,Padé 逼近式能给出关于函数值新的信息。这一点看起来很神秘。下面举例说明。

假定经过不懈努力,已求得某未知函数 $f(x)$ 的幂级数展开式的的前五项,

$$f(x) \approx 2 + \frac{1}{9}x + \frac{1}{81}x^2 - \frac{49}{8748}x^3 + \frac{175}{78732}x^4 + \cdots \tag{5.12.7}$$

(当然不必将系数写成精确的有理形式——写成数值的值就够了。这里写成有理形式是为了提醒读者这些系数是通过分析计算得到的。)图5.12.1中将式(5.12.7)描绘成标有"幂级数"的曲线。可以看出 $x\gtrsim4$ 时,主要是最大的四次方项支配着。

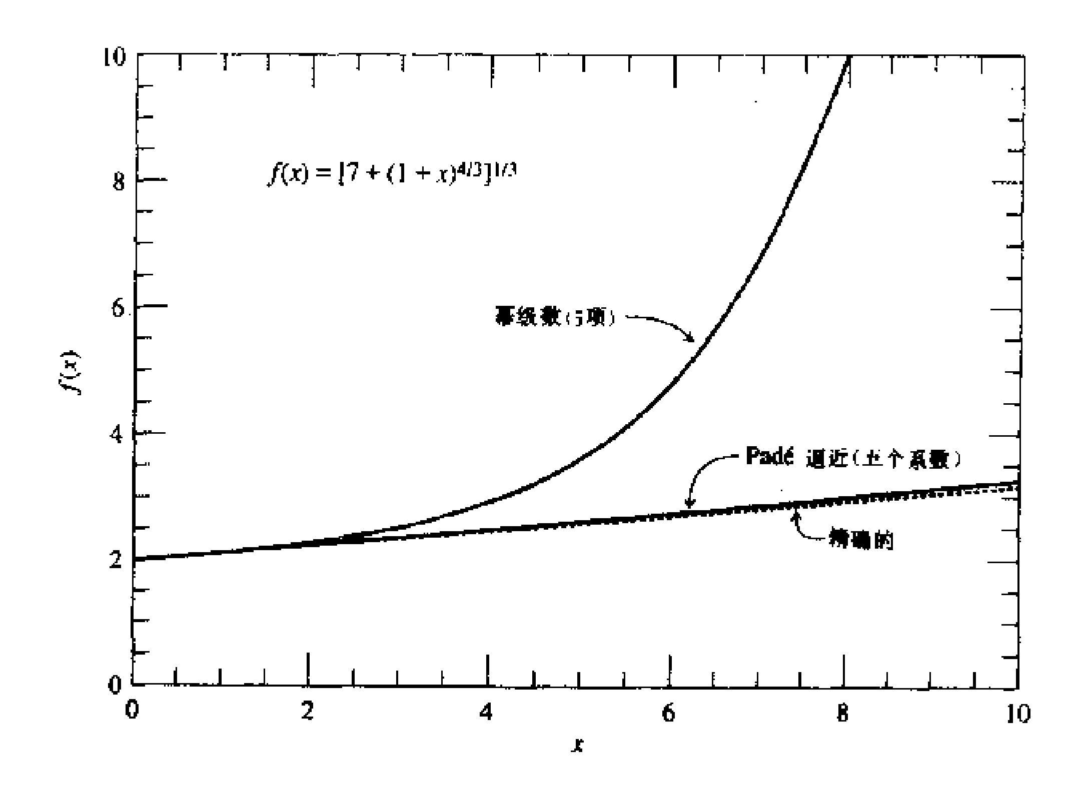
取式(5.12.7)中的五个系数,并输入给下面的程序 **Padé** 中运行。返回五个有理系数,3个 $a$ 值和2个 $b$ 值,它是将 $M=N=2$ 代入式(5.12.1)中得到。图5.12.1中标有"Padé"的曲线绘出所得到的有理函数。注意,图中两条实线都是由五个相同的原始系数值导出的。

为求得结果,我们需要用 Deus ex 机。结果告诉我们式(5.12.7)实际上就是下面函数的幂级数展开式

$$f(x) = [7 + (1 + x)^{4/3}]^{1/3} \tag{5.12.8}$$

见图中绘出的虚线。此函数在 $x=-1$ 处存在分枝点,故级数只在 $-1<x<1$ 范围内收敛。在图5.12.1中大部分区间,级数发散,因此对第五项的截尾项就没有意义了。但是,同样五项,转换成 Padé 逼近式后,得到的函数值至少在 $x\sim10$ 处仍有相当好的表示式。

为什么?有没有其它前五项数级幂相同的函数,但在 $2<x<10$ 范围内具有完全不同的性质?确实有。Padé 逼近式具有一种从想得到的各种可能性中挑选合适函数的神秘功能。Padé 逼近式的缺点是难于控制。一般无法确知其误差有多高,或者能有效地扩展到 $x$ 的哪个地方。这是一种有效的但仍是神秘的方法




幂级数仅对 $x<1$ 收敛。注意 Padé 逼近式在远远超过级数收敛半径的地方仍能保证精确性

图5.12.1　五项幂级数展开式及导出的同一函数 $f(x)$ 的五项系数 Padé 逼近式

下面程序从 $c$ 中计算出 $a$ 和 $b$。注意，程序是针对 $m=N$ 特殊情况，而且输出的有理系数排放成便于求值程序 **ratval**(第5.3节)使用的格式。(同样为保证程序兼容性，数组 $c$ 输入为双精度。)

```
#include <math.h>
#include "nrutil.h"
#define BIG 1.0e30

void pade(double cof[], int n, float *resid)
```
给定函数的幂级数展开式前几项系数 cof[0..2*n]，解线性 Padé 方程，返回同一函数的对角有理函数逼近式的系数，函数为(cof[0]+cof[1]$x$+...+cof[n]$x^N$)/(1+cof[n+1]$x$+...+cof[2*n]$x^N$)。值 resid 是剩余向量的范数；其值很小表明为收敛很好的解。注意，为保持与 **ratval** 兼容性 cof 是双精度。
```
{
    void lubksb(float **a, int n, int *indx, float b[]);
    void ludcmp(float **a, int n, int *indx, float *d);
    void mprove(float **a, float **alud, int n, int indx[], float b[],
        float x[]);
    int j,k,*indx;
    float d,rr,rrold,sum,**q,**qlu,*x,*y,*z;

    indx=ivector(1,n);
    q=matrix(1,n,1,n);
    qlu=matrix(1,n,1,n);
    x=vector(1,n);
    y=vector(1,n);
    z=vector(1,n);
```

```
    for (j=1;j<=n;j++) {                    为求解设置矩阵
        y[j]=x[j]=cof[n+j];
        for (k=1;k<=n;k++) {
            q[j][k]=cof[j-k+n];
            qlu[j][k]=q[j][k];
        }
    }
    ludcmp(qlu,n,indx,&d);                   由LU分解和回代法求解
    lubksb(qlu,n,indx,x);
    rr=BIG;
    for (;;) {                               使用迭代优化很重要，因为Padé方法有病态
        rrold=rr;
        for (j=1;j<=n;j++) z[j]=x[j];
        mprove(q,qlu,n,indx,y,x);
        for (rr=0.0,j=1;j<=n;j++)           计算剩余量
            rr += SQR(z[j]-x[j]);
        if (rr >= rrold) break;             若不能再优化，退出
    }
    *resid=sqrt(rr);
    for (k=1;k<=n;k++) {                     计算其它系数
        for (sum=cof[k],j=1;j<=k;j++) sum -= x[j]*cof[k-j];
        y[k]=sum;
    }                                        输出结果
    for (j=1;j<=n;j++) {
        cof[j]=y[j];
        cof[j+n] = -x[j];
    }
    free_vector(z,1,n);
    free_vector(y,1,n);
    free_vector(x,1,n);
    free_matrix(qlu,1,n,1,n);
    free_matrix(q,1,n,1,n);
    free_ivector(indx,1,n);
}
```

参考文献和进一步读物：


Ralston, A. and Wilf, H.S. 1960, *Mathematical Methods for Digital Computers*(New York: Wiley), p. 14.

Cuyt, A., and Wuytack, L. 1987, *Nonlinear Methods in Numerical Analysis* (Amsterdam: North-Holland), Chapter 2.

Graves-Morris, P.R. 1979, in *Pade Approximation and its Applications*, Lecture Notes in Mathematics, vol. 765, L. Wuytack, ed. (Berlin: Springer-Verlag). [1]


## 5.13 有理切比雪夫逼近

在第5.8节和第5.10节中我们已经知道如何对给定区间 $a\leqslant x\leqslant b$ 内的给定函数 $f(x)$，求出其较好的多项式逼近式。本节中我们对此工作作一般化推广，寻找有理函数的逼近式。这样做的原因是，对有些区间和有些函数，从最优有理函数逼近式能比具有相同系数个数的最优多项式逼近式得到更高的精度。必须权衡的是，求有理函数逼近式不如求多项式逼近求来得简单直接。正如我们已知的，由切比雪夫多项式求多项式的逼近式很方便。

设有理函数 $R(x)$的分子为 $m$ 次，分母为 $k$ 次。则有

$$R(x)=\frac{p_0+p_1x+\dots+p_mx^m}{1+q_1x+\dots+q_kx^k}\approx f(x)\quad(a\leq x\leq b)\tag{5.13.1}$$

要求的未知量是 $p_0,\dots,p_m$ 和 $q_1,\dots,q_k$ 即共有 $m+k+1$ 个量。设 $r(x)$表示 $R(x)$与 $f(x)$之差，且 $r$ 表示其绝对值的最大值

$$r(x)\equiv R(x)-f(x)\quad r\equiv\max_{a\leq x\leq b}|r(x)|\tag{5.13.2}$$

理想的最优解就是求适当 $p$、$q$ 之值使得 $r$ 最小。显然存在最小解，因为 $r$ 的下限为零。但如何去求解，或者如何合适地逼近它呢？

首先，可从下面基本定理获得一些启发：若 $R(x)$ 非退化（即分子、分母中没有相同的多项式因子），则使得 $r$ 最小的 $p$、$q$ 的值是唯一的；在选择此最优值时，$r(x)$ 在 $a \le x \le b$ 中有 $m+k+2$ 个极值，大小为 $r$ 且符号交替变化。（定理中省去了许多假设，准确描述见 Ralson[1]）。由此可知，有理函数情况与最小多项式情况非常相似；在第5.8节中已看到，具有 $n+1$ 个切比雪夫系数的 $n$ 阶逼近式的误差项，主要是由第一个被忽略的切比雪夫项构成，即 $T_{n+1}$，它本身有 $n+2$ 个幅度相等的极点，且符号交替变化。因此，这里有理系数的个数 $m+k+1$，与多项式系数的个数 $n+1$，所起的作用是相同的。

$r(x)$ 有 $m+k+2$ 个极值的另一种解释方法是，可使得 $R(x)$ 在任意 $m+k+1$ 个点 $x_i$ 处精确地等于 $f(x)$。对式(5.13.1)乘以它的分母得到方程

$$p_0 + p_1 x_i + \dots + p_m x_i^m = f(x_i)(1 + q_1 x_i + \dots + q_k x_i^k) \qquad i = 1,2,\dots,m+k+1 \tag{5.13.3}$$

这是一组未知量 $p$ 和 $q$ 的 $m+k+1$ 个线性方程组，可由标准方法（如 $LU$ 分解）求解。若选定所有 $x_i$ 值使其落在 $(a,b)$ 中，则在每个选定的 $x_i$ 和 $x_{i+1}$ 之间有一个极值点，再加上不在区间 $(a,b)$ 中的极值点，总共 $m+k+2$ 个极值点。对 $x_i$ 的任意极值并不总是有相同幅度。定理中表明选定一组特别的 $x_i$，幅度可能相同，且等于最小值 $r$。

若不用在 $x=x_i$ 处使 $f(x_i)$ 和 $R(x_i)$ 相等方法，也可以通过解下面线性方程使剩余量 $r(x_i)$ 等于任意设定值 $y_i$

$$p_0 + p_1 x_i + \dots + p_m x_i^m = [f(x_i) - y_i](1 + q_1 x_i + \dots + q_k x_i^k) \qquad i = 1,2,\dots,m+k+1 \tag{5.13.4}$$

事实上，若 $x_i$ 选定为最小解的极值点（非零点），则满足的方程为

$$p_0 + p_1 x_i + \dots + p_m x_i^m = [f(x_i) \pm r](1 + q_1 x_i + \dots + q_k x_i^k) \qquad i = 1,2,\dots,m+k+2 \tag{5.13.5}$$

其中 $\pm$ 符号随交替极值点而交替变化。注意式(5.13.5)在 $m+k+2$ 个极值点处都满足，而式(5.13.3)仅在 $m+k+1$ 个任意点处满足，为什么会这样呢？原因是式(5.13.5)中的 $r$ 也是未知值，故未知数共有 $m+k+2$ 个。方程组有点非线性（对 $r$ 而言），但用第九章介绍的方法还是可求解的。

这样，只要给定最优有理函数的极点位置，即可解得系数及最大偏差。另外有一条可导出的称为 **Remes 算法**[1]的定理，它讨论如何通过迭代过程收敛到这些位置。例如，下面是 **Remes 第二算法**的表述（稍微有些简化）：(1)求有 $m+k+2$ 个极点 $x_i$ 的初始有理函数（偏差不同）。(2)解式(5.15.5)求得新的有理系数及 $r$。(3)估算得到的 $R(x)$ 找出它的实际极点（与猜测值不一样）。(4)用具有相同符号最接近的极值点取代猜测值。(5)回到第二步并迭代直至收敛。在很多假定条件下，此方法收敛。Ralson[1]详细讨论细节，包括如何找 $x_i$ 的初始集。

迄今为止，我们的讨论仍然停留在纸上。现在让我们脱离常规，不用很漂亮的 Remes 算法。此算法内含的两种迭代（非线性集(5.13.5)的 $r$ 和新得到 $x_i$ 的集）要求过于苛刻，而且对于退化情况要求一套新的特殊逻辑设计。我们甚至愿意更加不合常理，我们怀疑采用寻找刚好最优且偏差相同的逼近式，与所付出的努力相比是否值得，除非有很少的一些人，他们的工作就是要找到最佳解，并将它置入编辑器和芯片中去。

当我们使用有理函数逼近时，目标通常非常实际：在某些内部循环中极其多次地反复求函数值，而我们只要加快求值速度。我们几乎从来不要求精确到机器精度的最后一位。假设我们使用的逼近式，其误差有 $m+k+2$ 个极点，偏差为因子 2。Remes 算法作为基础的定理保证：最优解的极点在因子 2 范围内[illegible]使上面极点下降，就会引起下面极点上升，直至都相等。这样，我们的不严格逼近式实际上至少在最优解的一位有效位的范围内。

这已经足够了，特别当寻找一种求不严格逼近式稳定的方法时，这就更好了。这种方法就是由[illegible]的

分解得到的超定线性方程的最小二乘方解（见第2.6节和15.4节）。按下面过程进行：首先，解（由最小二乘方）式(5.13.5)，不是求 $x_i$ 的 $m+k+1$ 个极值，而是求 $x_i$ 的非常大的值。它的空间分布跟高阶切比雪夫多项式零点分布相似。这就是 $R(x)$ 的初始估计。第二步，对结果的偏差列成表，求平均绝对偏差，称为 $r$。然后将 $r$ 固定，并对每点 $x_i$ 的偏差量的符号分别选定为 ±，以求解（再次在最小二乘方意义下）方程(5.13.5)。第三步，重复第二步几次。

我们的算法隐含有一些 Remes 的正统算法：求解方程不是将偏差置向零，而是加减一些一致性的值。然而我们没有跟踪实际极点，而只是每次求解线性方程。还有一个方法是，解加权最小二乘方问题，其中权的选择是使最大偏差迅速减小的。

下面是一段实现算法的程序。注意，只在初始填写表 fs 时才调用一次函数 fn。很容易将代码修改一下，以便在程序外面填写表。甚至坐标 xs 不必刚好就是我们用的那些，虽然，在最优解的每个极点之间坐标选得不够多时，会使拟合质量受到影响。注意，输出有理系数的格式适合于第5.3节程序 **ratval** 的调用。

```
#include <stdio.h>
#include <math.h>
#include "nrutil.h"
#define NPFAC 8
#define MAXIT 5
#define PIO2 (3.141592653589793/2.0)
#define BIG 1.0e30

void ratlsq(double (*fn)(double), double a, double b, int mm, int kk, double cof[], double *dev)
    以cof[0..mm+kk]返回函数fn在区间(a,b)的有理函数逼近系数。输入量mm和kk分别表示分子和分母的次数
    最大逼近偏差绝对值返回为dev。
{
    double ratval(double x, double cof[], int mm, int kk);
    void dsvbksb(double **u, double w[], double **v, int m, int n, double b[], double x[]);
    void dsvdcmp(double **a, int m, int n, double w[], double **v);     这些是svdcmp和svbksb的双
                                                                          精度函数
    int i,it,j,ncof,npt;
    double devmax,e,hth,power,sum,*bb,*coff,*ee,*fs,**u,**v,*w,*wt,*xs;

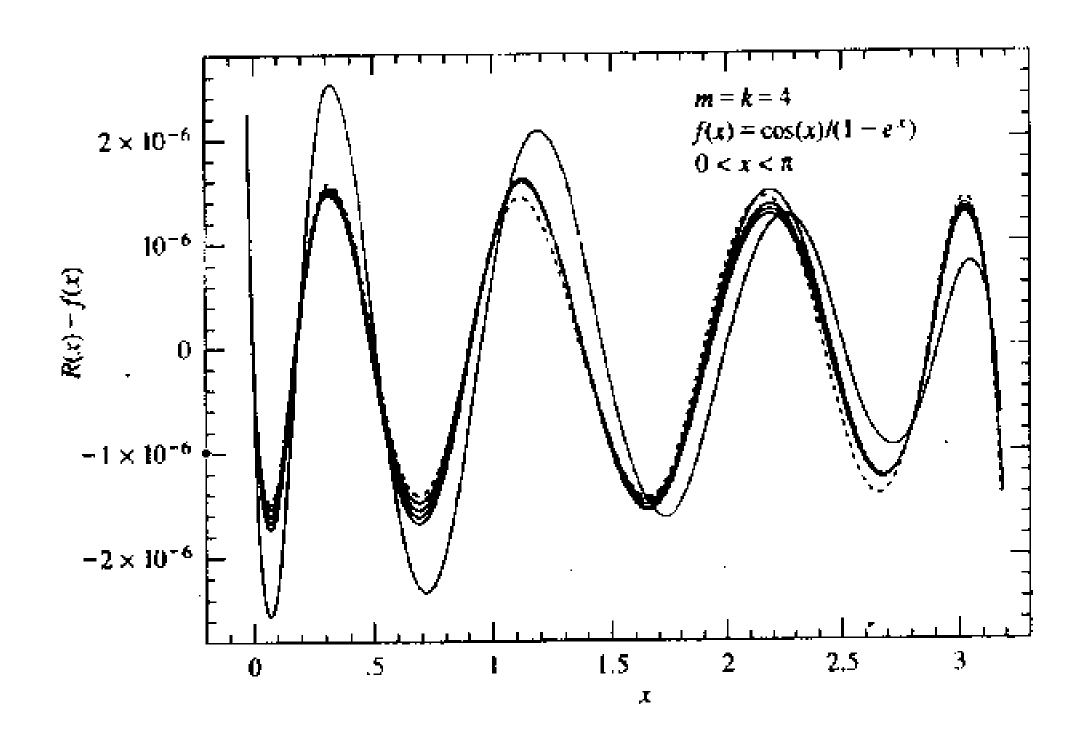
    ncof=mm+kk+1;
    npt=NPFAC*ncof;                                  函数值的点数，即网格密度
    bb=dvector(1,npt);
    coff=dvector(0,ncof-1);
    ee=dvector(1,npt);
    fs=dvector(1,npt);
    u=dmatrix(1,npt,1,ncof);
    v=dmatrix(1,ncof,1,ncof);
    w=dvector(1,ncof);
    wt=dvector(1,npt);
    xs=dvector(1,npt);
    *dev=BIG;
    for (i=1;i<=npt;i++) {                           数组装入求值点坐标和函数值
        if (i < npt/2) {
            hth=PIO2*(i-1)/(npt-1.0);                每次末尾，用公式使得舍入灵敏度最小
            xs[i]=a+(b-a)*DSQR(sin(hth));
        } else {
            hth=PIO2*(npt-i)/(npt-1.0);
            xs[i]=b-(b-a)*DSQR(sin(hth));
        }
        fs[i]=(*fn)(xs[i]);
        wt[i]=1.0;                                   将在迭代中调整数值以减少最大偏差
        ee[i]=1.0;
```

```
e=0.0;
for (it=1;it<=MAXIT;it++) {                              反复迭代
    for (i=1;i<=npt;i++) {                               为最小二乘方拟合设置"设计"矩阵
        power=wt[i];
        bb[i]=power*(fs[i]+SIGN(e,ee[i]));               这里是关键思想:偏离为正,拟合 $f_i(x)+e$;偏差为负,
                                                         拟合 $f_i(x)-e$。假设 e 是使渐近成为等波动偏差
        for (j=1;j<=mm+1;j++) {
            u[i][j]=power;
            power *= xs[i];
        }
        power = -bb[i];
        for(j=mm+2;j<=ncof;j++) {
            power *=xs[i];
            u[i][j]=power;
            }
    }
    dsvdcmp(u,npt,ncof,w,v);                             奇异值分解
    dsvbksb(u,w,v,npt,ncof,bb,coff-1);   在奇异或病难的情况下,这里修正奇异值 w[1..ncof],用零替
                        换最小值。注意,dsvbksb 函数中使用单位偏移数组,所以零偏移数组 coff 传入时应减1
    devmax=sum=0.0;
    for (j=1;j<=npt;j++) {                               对偏离值列表,修正数值
        ee[j]=ratval(xs[j],coff,mm,kk)-fs[j];
        wt[j]=fabs(ee[j]);                               用数值突出最偏离的点
        sum += wt[j];
        if (wt[j] > devmax) devmax=wt[j];
    }
    e=sum/npt;                                           将 e 更新平均绝对值偏离
    if (devmax <= *dev) {                                只存储找到的最好系数集
        for (j=0;j<ncof;j++) cof[j]=coff[j];
        *dev=devmax;
    }
    printf(" ratlsq iteration= %2d max error= %10.3e\n",it,devmax);
}
free_dvector(xs,1,npt);
free_dvector(wt,1,npt);
free_dvector(w,1,ncof);
free_dmatrix(v,1,ncof,1,ncof);
free_dmatrix(u,1,npt,1,ncof);
free_dvector(fs,1,npt);
free_dvector(ee,1,npt);
free_dvector(coff,0,ncof-1);
free_dvector(bb,1,npt);
}
```

图 5.13.1 描述了在区间$(0,\pi)$内求当$m=k=4$时,函数$f(x)=\cos x/(1+e^x)$的有理数拟合,用程序 **ratlsq** 迭代前五次的不同情况,可看出经过第一次迭代,结果已经差不多和最优解一样好,迭代收敛并不是如图形显示的那样:其实收敛最好的是第二次迭代(偏差最小)。程序 **ratlsq** 自然返回迭代最好的结果,并非就是最后一次,迭代超过五次并没有多大好处。

参考文献和进一步读物:


Ralston, A. and Wilf, H. S. 1960, *Mathematical Methods for Digital Computers* (New York: Wiley), Chapter 13. [1]




对于任意试验问题，实曲线显示程序 **ratlsq** 五次连续迭代的偏差 $r(x)$。这算法不能精确收敛到成小最大解(点曲线表示)。但是，第一次迭代后，这差异只是精度的最低有效位的很小部分。

图5.13.1

## 5.14 线积分求函数值

编写计算机程序时，选择的算法并非必须是效率最高，或实现最漂亮，或者是执行速度最快的。实际上，可能选择的是易于实现、通用而且便于检测的算法。

有时可能需要求几次或几千次某个特殊函数值，也许是复变量的复数函数，函数可能有很多不同参数或渐近区域，或者两种情况都存在。用通常的方法(级数、连分式、有理函数逼近，递推关系式等等)设计出的程序是，一系列条件测试以及对不同情况分支的大杂烩。程序执行时可能效率很高，但从一开始起通常不是最简便直接的方法。

一种极其通用但又不同的方法是，对一个函数定义的微分方程直接积分——对每个指定函数值从头开始积分——必要时，在复平面内沿一条路径积分。初看起来，这样做似乎是用金砖拍蚊子，大材小用。但现在情况是你已有了一块金砖，而一只蚊子正好就躺在下面，你要做的只要松手放掉金砖而已。

作为一个特例，考察复超几何函数 ${}_2F_1(a,b,c;z)$，根据超几何级数的解析连续性定义，

$$
{}_2F_1(a,b,c;z)=1+\frac{ab}{c}\frac{z}{1!}+\frac{a(a+1)b(b+1)}{c(c+1)}\frac{z^2}{2!}+\cdots
+\frac{a(a+1)\cdots(a+j-1)b(b+1)\cdots(b+j-1)}{c(c+1)\cdots(c+j-1)}\frac{z^j}{j!}+\cdots
\tag{5.14.1}
$$

级数仅在单园位 $|z|<1$ 内才收敛(见[1])，但我们感兴趣的不只是这个区域。

超几何函数 ${}_2F_1$ 是超几何差分方程，写为

$$z(1-z)F'' = abF - [c-(a+b+1)z]F' \tag{5.14.2}$$

的一个解（在原点处此解是规则的）。上式中撇号表示 $d/dz$。可以看出方程在 $z=0,1$ 和 $\infty$ 处有规则奇异点。因为所求解在 $z=0$ 处规则，这样 1 和 $\infty$ 成为分支点。若想使 ${}_2F_1$ 成为单值函数，则必须求一个联系两个点的切断分支。通常切断位置在正实轴从1到$\infty$之间，虽然某些别的场合下，可能需要另外的处理方法。

我们的“金砖”是，包括几个在后面第16章里将详细介绍的常微分方程组开始积分的程序集。现在只需一个高级“黑盒子”程序，从用独立变量值表示的初始条件开始积分，直到此变量用另一值表示的终止条件为止，并自动调整积分步长以保证结果精度。此程序为 **odeint**，程序中由复杂的 Bulisch-Stoer 方法计算步长。

假设已知某点 $z_0$ 处的函数 $F$ 及其导数 $F'$ 的值，现在需求函数 $F$ 在复平面的某个其它点处的值。连接两点的直线用参数表示为

$$z(s) = z_0 + s(z_1 - z_0) \tag{5.14.3}$$

其中 $s$ 为实参数。微分方程(5.14.2)可改写为两个一阶方程

$$\begin{aligned}\frac{dF}{ds} &= (z_1 - z_0)F' \\ \frac{dF'}{ds} &= (z_1 - z_0)\left(\frac{abF - [c-(a+b+1)z]F'}{z(1-z)}\right)\end{aligned} \tag{5.14.4}$$

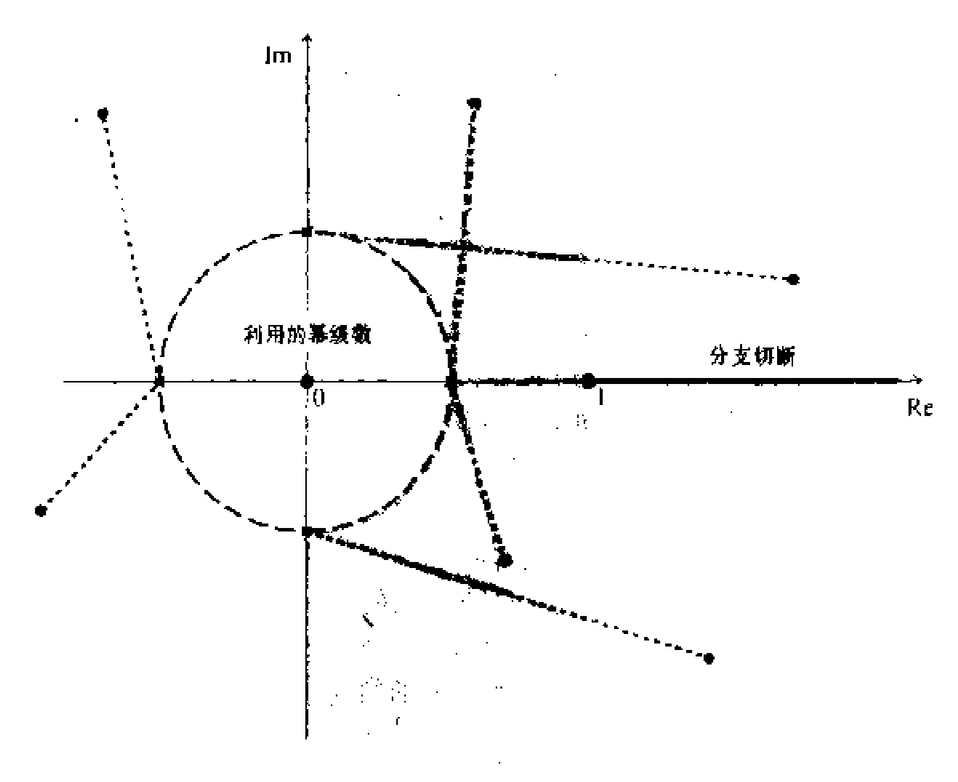
从 $s=0$ 到 $s=1$ 的积分。$F$ 和 $F'$ 被视为两个独立的复变量。撇号表示 $d/dz$ 这一点可以忽略，因为它可以呈现为式(5.14.4)中第一个方程的结果。此外，式(5.14.4)的实部和虚部决定了四个具有独立变量 $s$ 的实微分方程。式中右边的复数运算可视为耦合这四个部分的一种简便写法。程序 **odeint** 输入正是根据这种思想，因为它既不知道复数函数，也不知道复独立变量。

剩下的任务是确定从哪里开始，沿复平面内的哪条路径，达到任一点 $z$。此时，还要考虑函数奇点以及采用的分支切断点。图5.14.1表示了我们采用的策略。当 $|z|\leqslant\frac{1}{2}$ 时，式(5.14.1)的级数一般收敛很快，此时直接使用它就可以了。其它情况下，则沿从一个起点 $(\pm1/2,0)$ 或 $(0,\pm1/2)$ 开始的直线进行积分。选前一个起点分别对应于 $0<\mathrm{Re}(z)<1$ 及 $\mathrm{Re}(z)<0$ 的情况。选后一个起点则用于 $\mathrm{Re}(z)>1$，在切断分支点的上或下；从实轴开始的原因是为了避免太过于靠近 $z=1$ 的奇点（见图15.4.1）。分支切断点的位置是根据，所采用的策略在 $\mathrm{Re}(z)>1$ 时从不越过实轴进行积分，这一事实来确定的。

此算法的实现由第6.12节中的程序 **hypgeo** 给出。

基于上面描述过程的实施有各种衍生形式，都易于编程实现。若成功调用的 $z$ 值互相靠近（$a,b,c$ 值相同），则在每次调用时，可存储状态向量$(F,F')$及 $z$ 的相应值，然后在下一次调用时，用这些值作为起始值。增加的积分可能只需一至两步。应当避免无意中越过分支切断线进行积分；函数值可能是“正确的”，但不是所求的。

或者，也许想沿越过实轴 $\mathrm{Re}\ z>1$ 的折线路径作积分，把它作为一种移动分支切断点的方法。例如，若想从 $(0,1/2)$ 到 $(3/2,1/2)$ 积分，然后，从这里到任何 $\mathrm{Re}\ z>1$ 的地方——$\mathrm{Im}\ z$ 的符号正负均可。（例如，若想用迭代方法求一个函数的根，而不想在分支点处沿不同路径的附近值求积分。若正是这样，在求根时会发现不连续函数值，而且不收敛！）




表示超几何函数奇异点、分支切断点，以及几条虚线是从圆 $|z|=\frac{1}{2}$（这里级数收敛很快）到其它点几条积分路径的复平面。

图5.14.1

在复平面上，通过对微分方程积分来求函数值的方法，也可用于其它特殊函数。例如，复贝塞尔函数，艾瑞（Airg）函数，库劳姆泊（Coulomb）波形函数以及韦伯（Weber）函数，这些都是**合流超几何函数**的特例，其微分方程与上面使用过的类似（见参考书[1]中第13.6节的特例表）。合流超几何函数在有限 $z$ 值处没有奇异点，这样积分就容易些。但是，无穷远处的本质奇异性，意味着沿某些路径对某些参数，积分具有很强的振荡性或指数衰减性，这样积分就更困难了。所以有时候只有分不同情况讨论（或试探）。

参考文献和进一步读物：


Abramowitz, M., and Stegun, I. A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, vol. 55. [1]

# 第六章　特殊函数

## 6.0　引言

**特殊函数**并没有什么特殊的地方，只是因为某些权威人士或教科书的作者这么称呼的。特殊函数又称为**高等超越函数**（比什么东西高?），或称**数学物理函数**（但它们也经常出现在其它领域里），或称**满足某种频繁发生的二阶微分方程**的函数（但并不是所有特殊函数都如此）。或许多人简单地称为"有用函数"而不作深究。本章内容应包括哪些函数，这的确只是一个尝试性的问题。

一些商品化了的很好的软件包，如 NAG 或 IMSL 就包含许多特殊函数的程序。这种程序只对那些不想知道其内部运行情况的用户是很有用的。通常这种黑箱里面状态杂乱，程序中分枝很多，以调用参数的不同，所使用方法也不完全不同。黑箱能够（而且也应该能够）仔细地控制精度，以便在各种条件状况下达到所要求的精度。我们给出的示例将不太讲究这些，一则因为我们想阐述第五章的方法，二则因为我们想让读者能理解所提供程序的内部运行情况。一些程序中含有精度参数，可按要求设置其大小，其它一些程序中（尤其是多项式拟合时），只给出某种一般可行（通常6位左右有效数字）的精度。我们不能保证程序黑箱是完美的。只希望读者在程序中遇到问题时，能根据我们提供的材料，诊断并能修正问题。

总之，本章的特殊函数子程序可直接使用——我们也是一直在用——但我们也希望用户能了解其内部工作情况。

## 6.1　Γ函数，β函数，阶乘，二项式系数

Γ函数（*Gamma* 函数）定义为积分形式

$$\Gamma(z) = \int_0^\infty t^{z-1}e^{-t}dt \qquad (6.1.1)$$

$z$ 若为整数，Γ函数就是熟知的阶乘，但必须偏移1，即

$$n! = \Gamma(n+1) \qquad (6.1.2)$$

Γ函数满足递推关系式

$$\Gamma(z+1) = z\Gamma(z) \qquad (6.1.3)$$

若参数 $z>1$，或者更一般地在复半平面 $\mathrm{Re}(z)>1$ 上函数已知，则可由反射公式得到 $z<1$ 或 $\mathrm{Re}(z)<1$ 时

$$\Gamma(1-z) = \frac{\pi}{\Gamma(z)\sin(\pi z)} = \frac{\pi z}{\Gamma(1+z)\sin(\pi z)} \qquad (6.1.4)$$

注意 $\Gamma(z)$ 在 $z=0$ 及所有负整数值处有极点。

计算函数 $\Gamma(z)$ 的数值方法有多种，但都不如由兰科斯（Lanczos）导出的近似公式来得干净利落[1]。它只对Γ函数完全有效，似乎是无中生有的。这里不打算去导出近似公式，只给

出其结果；对某一选定的 $\gamma$ 和 $N$，及一组系数 $c_1, c_2, \cdots, c_N$，$\Gamma$ 函数由下式给出

$$\Gamma(z+1) = \left(z+\gamma+\frac{1}{2}\right)^{z+\frac{1}{2}} e^{-\left(z+\gamma+\frac{1}{2}\right)}$$

$$\times \sqrt{2\pi}\left[c_0 + \frac{c_1}{z+1} + \frac{c_2}{z+2} + \cdots + \frac{c_N}{z+N} + \epsilon\right] \quad (z>0) \tag{6.1.5}$$

可以看出上式思路类似于斯特林(Stirling)近似公式，只是考察到左半复平面的一些一阶极点而作了一些修正。常数 $c_0$ 很显然应等于1。误差项用 $\epsilon$ 表示。对 $\gamma=5$，$N=6$，及一组 $c$ 值，误差项 $|\epsilon|<2\times10^{-10}$。有印象了吗？若还没有，读者一定会对下面事实留下深刻印象的：即（在相同参数下）公式(6.1.5)和 $\epsilon$ 的上下限在 $\mathrm{Re}(z)>0$ 的复半平面的任何地方同样适用于复 $\Gamma$ 函数。

计算 $\ln\Gamma(x)$ 比 $\Gamma(x)$ 更好，因为后者在许多计算机浮点表示中，对一般 $x$ 值也会发生溢出。通常凡计算中用到 $\Gamma$ 函数处，一般总是用 $\Gamma(x)$ 去除以另一个很大的数，这样所得结果就是普通大小的值了。这种运算可转化为对数减法运算。有了式(6.1.5)，我们只须调用 2 次对数函数，和 25 次左右的算术运算，立即可得到 $\Gamma$ 函数对数值。这并不比我们想像的某些函数，如 $e^x$ 或 $\sin(x)$ 的计算困难多少。

```
#include <math.h>

float gammln(float xx)        当xx>0时返回值ln[Γ(xx)]。
{
    double x,y,tmp,ser;                 内部运算将用双精度，若五位数字的精确度已足够好，那么双精度忽略
    static double cof[6]={76.18009172947146,-86.50532032941677,
        24.01409824083091,-1.231739572450155,
        0.1208650973866179e-2,-0.5395239384953e-5};
    int j;

    y=x=xx;
    tmp=x+5.5;
    tmp -= (x+0.5)*log(tmp);
    ser=1.000000000190015;
    for (j=0;j<=5;j++) ser += cof[j]/++y;
    return -tmp+log(2.5066282746310005*ser/x);
}
```

怎样编写求阶乘函数 $n!$ 的程序呢？一般情况下，小整数去调用阶乘函数（数值大后总是溢出）；多数应用情况下，同一个整数值需多次调用。若每次需要阶乘都调用 **exp(gammln**($n$**+1.0))**，这样就太浪费机时了，最好返本求源，非必要时不要调用 **gammln**。

```
#include <math.h>

float factrl(int n)        按浮点数返回n!
{
    float gammln(float xx);
    void nrerror(char error_text[]);
    static int ntop=4;
    static float a[33]={1.0,1.0,2.0,6.0,24.0};        仅按要求填表
    int j;

    if (n < 0) nrerror("Negative factorial in routine factrl");
    if (n > 32) return exp(gammln(n+1.0));        需要大于表长的数，事实上多数计算机上会比大数一般
```

```
    while (ntop<n) {                          会溢出,但不妨一试
        j=ntop--;                             填表直到设置的值
        a[ntop]=a[j]*ntop;
    }
    return a[n];
}
```

$n$ 值较小时,**factrl** 结果将是精确的,因为在任何计算机上,浮点数乘以一个小整数的结果总是精确的。若要用阶乘对数,则这种精确性就不能保证了,但对于二项式系数,必须精确地求得阶乘,因为在二项式系数本身溢出以前,二项式中的阶乘早就已溢出了。

二项式系数定义为

$$\binom{n}{k}=\frac{n!}{k!(n-k)!} \qquad 0\leqslant k\leqslant n \tag{6.1.6}$$

```
#include <math.h>

float bico(int n, int k)      按浮点数返回二项式系数 (n k)
{
    float factln(int n);

    return floor(0.5+exp(factln(n)-factln(k)-factln(n-k)));     n,k 较小时,floor 函数清除舍入误差
}
```

使用

```
float factln(int n)           返回 ln(n!)
{
    float gammln(float xx);
    void nrerror(char error_text[]);
    static float a[101];                  静态数组自动初始化为零

    if (n < 0) nrerror("Negative factorial in routine factln");
    if (n <= 1) return 0.0;
    if (n <= 100) return a[n] ? a[n] : (a[n]=gammln(n+1.0));     在表内
else return gammln(n+1.0);                                          超出表内
}
```

求解问题时常需用到二项式系数,比较好的方法是利用递推公式,例如

$$\binom{n+1}{k}=\frac{n+1}{n-k+1}\binom{n}{k}=\binom{n}{k}+\binom{n}{k-1}$$

$$\binom{n}{k-1}=\frac{n-k}{k+1}\binom{n}{k} \tag{6.1.7}$$

暂时放下整值变量的组合函数,最后我们讨论一下 Beta 函数(B 函数)

$$B(z,w)=B(w,z)=\int_0^1 t^{z-1}(1-t)^{w-1}dt \tag{6.1.8}$$

与 Γ 函数的关系为

$$B(z,w)=\frac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)} \tag{6.1.9}$$

因此

```
#include <math.h>

float beta(float z, float w)    返回 B 函数值 B(z,w)
{
    float gammln(float xx);
    return exp(gammln(z)-gammln(w)-gammln(z-w));
}
```

参考文献和进一步读物：


Lanczos, C. 1964, *SIAM Journal on Numerical Analysis*, ser. B, vol. 1, pp. 86~96. [1]


## 6.2 不完全 Γ 函数、误差函数、$\chi^2$ 概率函数、累积泊松函数

不完全 Γ 函数定义为

$$P(a,x) = \frac{\gamma(a,x)}{\Gamma(a)} \equiv \frac{1}{\Gamma(a)}\int_0^x e^{-t}t^{a-1}dt \quad (a > 0) \tag{6.2.1}$$

它有极限值

$$P(a,0) = 0 \quad 和 \quad P(a,\infty) = 1 \tag{6.2.2}$$

不完全 Γ 函数 $P(a,x)$ 是单调的，当 $a$ 大于 1 左右时，在大约以 $a-1$ 为中心，宽为 $\sqrt{a}$ 的区域内，从"接近零"上升到"接近1"(见图6.2.1)。

$P(a,x)$ 的互补形式也统称为不完全 Γ 函数，

$$Q(a,x) \equiv 1 - P(a,x) = \frac{\Gamma(a,x)}{\Gamma(a)} \equiv \frac{1}{\Gamma(a)}\int_x^\infty e^{-t}t^{a-1}dt \quad (a > 0) \tag{6.2.3}$$

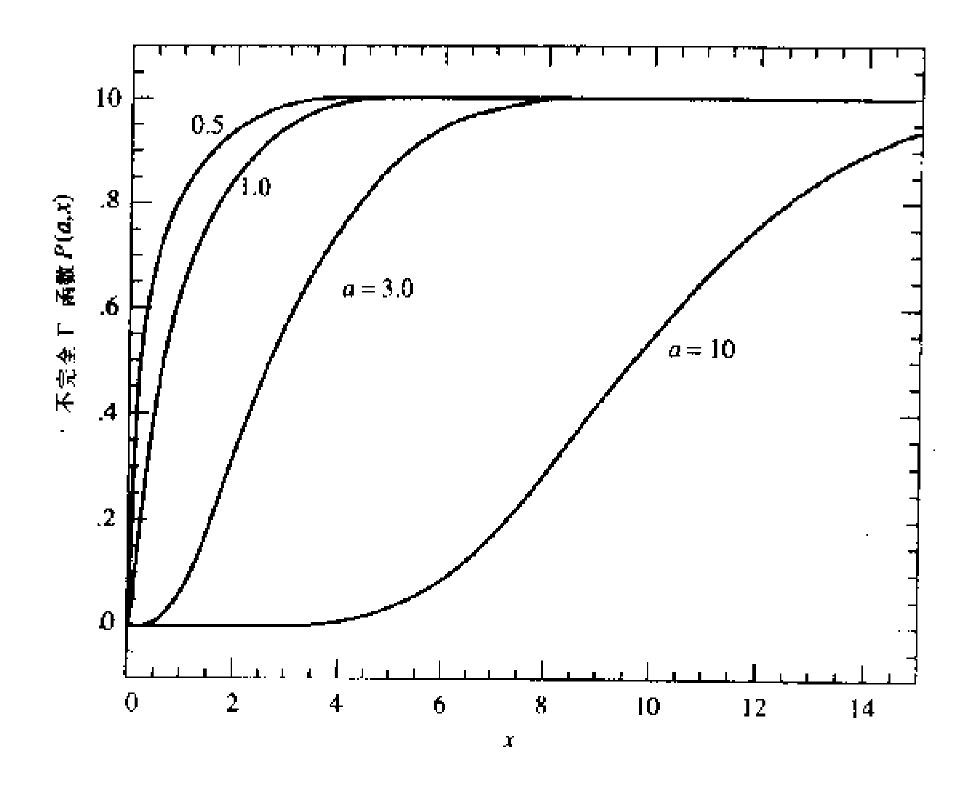



图6.2.1 四个 $a$ 值的不完全 Gamma 函数 $P(a,x)$

其极限值为

$$Q(a,0)=1 \quad 和 \quad Q(a,\infty)=0 \tag{6.2.4}$$

记号 $P(a,x)$，$\gamma(a,x)$和$\Gamma(a,x)$是标准记号，而记号$Q(a,x)$是本节专用记号。

$\gamma(a,x)$有如下级数展开式

$$\gamma(a,x)=e^{-x}x^a\sum_{n=0}^{\infty}\frac{\Gamma(a)}{\Gamma(a+1+n)}x^n \tag{6.2.5}$$

不必每次对 $n$ 重新计算一次 $\Gamma(a-1+n)$，最好利用式(6.1.3)及以前的系数。

$\Gamma(a,x)$的连分式展开为

$$\Gamma(a,x)=e^{-x}x^a\left(\frac{1}{x+}\ \frac{1-a}{1+}\ \frac{1}{x+}\ \frac{2-a}{1+}\ \frac{2}{x+}\cdots\right) \quad (x>0) \tag{6.2.6}$$

用式(6.2.6)偶项部分计算较好，它收敛快二倍(见第5.2节)

$$\Gamma(a,x)=e^{-x}x^a\left(\frac{1}{x+1-a-}\ \frac{1\cdot(1-a)}{x+3-a-}\ \frac{2\cdot(2-a)}{x+5-a-}\cdots\right) \tag{6.2.7}$$

可以发现式(6.2.5)当 $x$ 小于 $a+1$时收敛很快，而式(6.2.6)或(6.2.7)，当 $x$ 大于 $a+1$ 时收敛很快。每种情况下，均须到几倍 $\sqrt{a}$ 的项数后才收敛。在 $x=a$ 附近，不完全 $\Gamma$ 函数变化最剧烈。因此，式(6.2.5)及式(6.2.7)一起可对任何数 $a$ 及 $x$ 求函数值。附加的收益就是，我们再也不必通过两个几乎相等的数相减来求零附近的函数值。返回 $P(a,x)$和 $Q(a,x)$的高阶程序是

```
float gammp(float a, float x)        返回不完全Γ函数P(a,x)值.
{
    void gcf(float *gammcf, float a, float x, float *gln);
    void gser(float *gamser, float a, float x, float *gln);
    void nrerror(char error_text[]);
    float gamser,gammcf,gln;

    if (x < 0.0 || a <= 0.0) nrerror("Invalid arguments in routine gammp");
    if (x < (a+1.0)) {                   用级数表示
        gser(&gamser,a,x,&gln);
        return gamser;
    } else {
        gcf(&gammcf,a,x,&gln);            取互补形式
        return 1.0-gammcf;               用连分数表示
    }
}

float gammq(float a, float x)        返回不完全Γ函数Q(a,x)≡1-P(a,x)
{
    void gcf(float *gammcf, float a, float x, float *gln);
    void gser(float *gamser, float a, float x, float *gln);
    void nrerror(char error_text[]);
    float gamser,gammcf,gln;

    if (x < 0.0 || a <= 0.0) nrerror("Invalid arguments in routine gammq");
    if (x < (a+1.0)) {                      用级数表示
        gser(&gamser,a,x,&gln);
        return 1.0-gamser;                  取互补形式
    } else {                                用连分数表示
        gcf(&gammcf,a,x,&gln);
        return gammcf;
```

}

级数和连分式都将变量 gln 置为 $\ln\Gamma(a)$，原因是使用户方便地将上面两个程序修改成求 $\gamma(a,x)$ 和 $\Gamma(a,x)$，而不只是求 $P(a,x)$ 和 $Q(a,x)$。(参见式(6.2.1)和(6.2.3))。

实现式(6.2.5)及(6.2.7)的函数 gser、gcf 为

```
#include <math.h>
#define ITMAX 100
#define EPS 3.0e-7

void gser(float *gamser, float a, float x, float *gln)
    用级数表示式如 gamser 求不完全 Γ 函数值,并返回。同时由 gln 返回 lnΓ(a)。
{
    float gammln(float xx);
    void nrerror(char error_text[]);
    int n;
    float sum,del,ap;

    *gln=gammln(a);
    if (x <= 0.0) {
        if (x < 0.0) nrerror("x less than 0 in routine gser");
        *gamser=0.0;
        return;
    } else {
        ap=a;
        del=sum=1.0/a;
        for (n=1;n<=ITMAX;n++) {
            ++ap;
            del *= x/ap;
            sum += del;
            if (fabs(del) < fabs(sum)*EPS) {
                *gamser=sum*exp(-x+a*log(x)-(*gln));
                return;
            }
        }
        nrerror("a too large, ITMAX too small in routine gser");
        return;
    }
}


#include <math.h>
#define ITMAX 100              最大允许迭代次数
#define EPS 3.0e-7             相对精度
#define FPMIN 1.0e-30          最小可表示的浮点数的邻近数

void gcf(float *gammcf, float a, float x, float *gln)
    由连分数表示形式 gammcf 求不完全 Gamma 函数值并返回。同时由 gln 返回 Γ(a)。
{
    float gammln(float xx);
    void nrerror(char error_text[]);
    int i;
    float an,b,c,d,del,h;

    *gln=gammln(a);
    b=x+1.0-a;                                这些设置为了用 b0 = 0 的修正Lentz方法
    c=1.0/FPMIN;                                  (§5.2)来计算连分式
    d=1.0/b;
    h=d;
    for (i=1;i<=ITMAX;i++) {                  迭代到收敛
        an = -i*(i-a);
        b += 2.0;
```

```
        d=an*d+b;
        if (fabs(d) < FPMIN) d=FPMIN;
        c=b+an/c;
        if (fabs(c) < FPMIN) c=FPMIN;
        d=1.0/d;
        del=d*c;
        h *= del;
        if (fabs(del-1.0) < EPS) break;
    }
    if (i > ITMAX) nrerror("a too large, ITMAX too small in gcf");
    *gammcf=exp(-x+a*log(x)-(*gln))*h;          将因子放在前面
}
```

### 6.2.1 误差函数

误差函数及其互补形式是不完全Γ函数的特例，故可由前述过程方便地推得。其定义式为

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \tag{6.2.8}$$

及

$$\mathrm{erfc}(x) \equiv 1 - \mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt \tag{6.2.9}$$

它们有如下极限值和对称性质：

$$\mathrm{erf}(0) = 0 \quad \mathrm{erf}(\infty) = 1 \quad \mathrm{erf}(-x) = -\mathrm{erf}(x) \tag{6.2.10}$$

$$\mathrm{erfc}(0) = 1 \quad \mathrm{erfc}(\infty) = 0 \quad \mathrm{erfc}(-x) = 2 - \mathrm{erfc}(x) \tag{6.2.11}$$

与不完全Γ函数的关系为

$$\mathrm{erf}(x) = P(\frac{1}{2}, x^2) \qquad (x \geqslant 0) \tag{6.2.12}$$

和

$$\mathrm{erfc}(x) = Q(\frac{1}{2}, x^2) \qquad (x \geqslant 0) \tag{6.2.13}$$

为了避免与C语言库中函数名冲突，所以我们的程序名中多加进一个"f"。

```
float erff(float x)        返回误差函数erf(x)值
{
    float gammp(float a, float x);

    return x < 0.0 ? -gammp(0.5,x*x) : gammp(0.5,x*x);
}

float erffc(float x)       返回互补误差函数erfc(x)值
{
    float gammp(float a, float x);
    float gammq(float a, float x);

    return x < 0.0 ?1.0+gammp(0.5,x*x) : gammq(0.5,x*x);
```

如果愿意，还可很容易补救 **erff** 和 **erffc** 中一个小小的效率差之处，即调用 **gammp** 和 **gammq** 后，$\Gamma(0.5)=\sqrt{\pi}$ 也就不必计算了。首先，需要考虑下面一段程序，它是基于对函数形式猜测的切比雪夫拟合。

```
#include <math.h>
float erfcc(float x)      返回互补误差函数erfc(x),其相对误差处处小于1.2×10^-7
{
    float t,z,ans;

    z=fabs(x);
    t=1.0/(1.0+0.5*z);
    ans=t*exp(-z*z-1.26551223+t*(1.00002368+t*(0.37409196+t*(0.09678418+
        t*(-0.18628806+t*(0.27886807+t*(-1.13520398+t*(1.48851587+
        t*(-0.82215223+t*0.17087277)))))))));
    return x>=0.0 ? ans : 2.0-ans;
}
```

还有一些不完全Γ函数的特例,它们带有两个变量。

### 6.2.2 累积泊松(Possion)概率函数

$P_x(<k)$表示**累积泊松概率函数**,其中$x$为正数,整数$k\geq 1$。若期望均值为$x$,则此函数定义为泊松随机事件发生次数介于0到$k-1$(包括0和$k-1$在内)的概率。其极限值为

$$P_x(<1)=e^{-x} \qquad P_x(<\infty)=1 \tag{6.2.14}$$

与不完全Γ函数简单关系为

$$P_x(<k)=Q(k,x)=\mathbf{gammq}(k,x) \tag{6.2.15}$$

### 6.2.3 $\chi^2$ 概率函数

$P(\chi^2|\nu)$定义为正确模型下观察$chi$平方值小于值$\chi^2$的概率(我们将在第十五章讨论这一函数的用处)。其互补$Q(\chi^2|\nu)$为正确模型下观察chi平方值随机超过$\chi^2$值的概率。两式中$\nu$为整数,表示自由度数。函数有极限值

$$P(0|\nu)=0 \qquad P(\infty|\nu)=1 \tag{6.2.16}$$

$$Q(0|\nu)=1 \qquad Q(\infty|\nu)=0 \tag{6.2.17}$$

与不完全Γ函数有如下关系:

$$P(\chi^2|\nu)=P\left(\frac{\nu}{2},\frac{\chi^2}{2}\right)=\mathbf{gammp}\left(\frac{\nu}{2},\frac{\chi^2}{2}\right) \tag{6.2.18}$$

$$Q(\chi^2|\nu)=Q\left(\frac{\nu}{2},\frac{\chi^2}{2}\right)=\mathbf{gammp}\left(\frac{\nu}{2},\frac{\chi^2}{2}\right) \tag{6.2.19}$$

## 6.3 指数积分

指数积分的标准定义为:

$$E_n(x)=\int_1^{\infty}\frac{e^{-xt}}{t^n}dt \qquad x>0,\ n=0,1,2,\ldots \tag{6.3.1}$$

下面用积分的主值定义的函数

$$\mathrm{Ei}(x)=-\int_{-x}^{\infty}\frac{e^{-t}}{t}\,dt=\int_{-\infty}^{x}\frac{e^t}{t}\,dt \qquad x>0 \tag{6.3.2}$$

也称为指数积分。注意，由于解析连续性，$\mathrm{Ei}(-x)$等于$-E_1(x)$。

函数 $E_n(x)$是不完全 Γ 函数的特例：

$$E_n(x) = x^{n-1}\Gamma(1-n, x) \tag{6.3.3}$$

这样可用类似的方法求其值。整理(6.2.6)式写成连分式形式，对所有 $x>0$ 都收敛：

$$E_n(x) = e^{-x}\left(\frac{1}{x+}\ \frac{n}{1+}\ \frac{1}{x+}\ \frac{n+1}{1+}\ \frac{2}{x+}\cdots\right) \tag{6.3.4}$$

它的偶项形式收敛更快：

$$E_n(x) = e^{-x}\left(\frac{1}{x+n-}\ \frac{1\cdot n}{x+n+2-}\ \frac{2(n+1)}{x+n+4-}\cdots\right) \tag{6.3.5}$$

仅当 $x\gtrsim 1$ 时，连分式形式才很快收敛，这时才有实用价值。对 $0<x\lesssim 1$，则用级数表示

$$E_n(x) = \frac{(-x)^{n-1}}{(n-1)!}[-\ln x + \Psi(n)] - \sum_{\substack{m=0\\ m\neq n-1}}^{\infty}\frac{(-x)^m}{(m-n+1)m!} \tag{6.3.6}$$

参量 $\Psi(n)$为双 Γ 函数，对整数变量定义为：

$$\Psi(1) = -\gamma, \quad \Psi(n) = -\gamma + \sum_{m=1}^{n-1}\frac{1}{m} \tag{6.3.7}$$

其中 $\gamma=0.5772156649\cdots$，即欧拉(Euler)常数。按 $x$ 的幂次递增顺序求式(6.3.6)的值：

$$\begin{aligned}E_n(x) = &-\left[\frac{1}{(1-n)} - \frac{x}{(2-n)\cdot 1} + \frac{x^2}{(3-n)(1\cdot 2)} - \cdots + \frac{(-x)^{n-2}}{(-1)(n-2)!}\right]\\ &+ \frac{(-x)^{n-1}}{(n-1)!}[-\ln x + \Psi(n)] - \left[\frac{(-x)^n}{1\cdot n!} + \frac{(-x)^{n+1}}{2\cdot(n+1)!} + \cdots\right]\end{aligned} \tag{6.3.8}$$

$n=1$ 时则省去第一个方括号。这种方法的优点是，$n$ 很大时级数项在到达含 $\Psi(n)$项之前就开始收敛。这样就只须对较小的 $n$ 如 $20< n <40$ 找一个算法求 $\Psi(n)$。虽然查找表方法效率稍高些，但我们仍用式(6.3.7)。

Amos[1]详细讨论了求式(6.3.8)时截尾误差的影响，并且给出一个相当精巧的结束条件。但我们发现，若仅仅在加的最后一项小于允许值时停止，效果也不错。

两种特殊情况需分别处理：

$$\begin{aligned}E_0(x) &= \frac{e^{-x}}{x}\\ E_n(0) &= \frac{1}{n-1}, \qquad n>1\end{aligned} \tag{6.3.9}$$

下面给出的程序函数 **expint**，它允许以精度 EPS 快速计算 $E_n(x)$，EPS 达到机器浮点数允许的字长范围内。若仍想更精确的结果，则只能在欧拉常数后面再增添几位数字了。Wrehch[2]给出了前 328 位数字。

```
#include <math.h>
#define MAXIT 100                    允许最大迭代数
#define EULER 0.5772156649           欧拉常数 γ
#define FPMIN 1.0e-30                接近最小可表示的浮点数
#define EPS 1.0e-7                   期望的相对误差不小于机器精度

float expint(int n, float x)     求指数积分 En(x)
```

```
{
    void nrerror(char error_text[]);
    int i,ii,nm1;
    float a,b,c,d,del,fact,h,psi,ans;

    nm1=n-1;
    if (n < 0 || x < 0.0 || (x==0.0 && (n==0 || n==1)))
    nrerror("bad arguments in expint");
    else {
        if (n == 0) ans=exp(-x)/x;                    特殊情况
        else {
            if (x == 0.0) ans=1.0/nm1;                另一种特殊情况

            else {
                if (x > 1.0) {                         Lentz 算法(§5.2)
                    b=x+n;
                    c=1.0/FPMIN;
                    d=1.0/b;
                    h=d;
                    for (i=1;i<=MAXIT;i++) {
                        a = -i*(nm1+i);
                        b += 2.0;
                        d=1.0/(a*d+b);                 分母不能为零
                        c=b+a/c;
                        del=c*d;
                        h *= del;
                        if (fabs(del-1.0) < EPS) {
                            ans=h*exp(-x);
                            return ans;
                        }
                    }
                    nrerror("continued fraction failed in expint");
                } else {                               求级数值
                    ans = (nm1!=0 ? 1.0/nm1 : -log(x)-EULER);       设置前几项
                    fact=1.0;
                    for (i=1;i<=MAXIT;i++) {
                        fact *= -x/i;
                        if (i != nm1) del = -fact/(i-nm1);
                        else {
                            psi = -EULER;              计算 ψ(n)
                            for (ii=1;ii<=nm1;ii++) psi += 1.0/ii;
                            del=fact*(-log(x)+psi);
                        }
                        ans += del;
                        if (fabs(del) < fabs(ans)*EPS) return ans;
                    }
                    nrerror("series failed in expint");
                }
            }
        }
    }
    return ans;
}
```

求 Ei 值较好的方法是，$x$ 比较小时用幂级数，而 $x$ 较大时用渐近级数。幂级数为：

$$\mathrm{Ei}(x) = \gamma + \ln x + \frac{x}{1 \cdot 1!} + \frac{x^2}{2 \cdot 2!} + \dots \tag{6.3.10}$$

其中 $\gamma$ 为欧拉常数。渐近级数是：

$$\mathrm{Ei}(x) \sim \frac{e^x}{x}\left(1 + \frac{1!}{x} + \frac{2!}{x^2} + \dots\right) \tag{6.3.11}$$

使用渐近级数的下限 $x$ 大约为 $|\ln \mathrm{EPS}|$，EPS 是要求的相对误差。

```
#include <math.h>
#define EULER 0.57721566              欧拉常数γ
#define MAXIT 100                     允许的最大迭代数
#define FPMIN 1.0e-30                 接近最小可表示的浮点数
#define EPS 6.0e-8                    相对误差，或者在x=0.3725处Ei接近零的绝对误差

float ei(float x)       x>0时计算指数积分Ei(x)
{
    void nrerror(char error_text[]);
    int k;
    float fact,prev,sum,term;

    if (x <= 0.0) nrerror("Bad argument in ei");
    if (x < FPMIN) return log(x)+EULER;          特殊情况：防止因为下溢导致收敛判断失败
    if (x <= -log(EPS)) {
        sum=0.0;                                 用幂级数
        fact=1.0;
        for (k=1;k<=MAXIT;k++) {
            fact *= x/k;
            term=fact/k;
            sum += term;
            if (term < EPS*sum) break;
        }
        if (k > MAXIT) nrerror("Series failed in ei");
        return sum+log(x)+EULER;
    } else {                                     用渐近级数
        sum=0.0;                                 从第二项开始
        term=1.0;
        for (k=1;k<=MAXIT;k++) {
            prev=term;
            term *= k/x;
            if (term < EPS) break;
            因为最后一项大于1,term本身约等于EPS
            if (term < prev) sum += term;        还收敛：加入新的D项
            else {
                sum -= prev;                     发散：减去前一项并退出
                break;
            }
        }
        return exp(x)*(1.0+sum)/x;
    }
}
```

参考文献和进一步读物：


Stegun, I. A., and Zucker, R. 1974, *Journal of Research of the National Bureau of Standards*, vol. 78B, pp. 199～216; 1976, *op. cit.*, vol. 80B, pp. 291～311.

Amos D. E. 1980, *ACM Transactions on Mathematical Software*, vol. 6, pp. 365～377 [1]; also vol. 6, pp. 420～428.

Wrench J. W. 1952, *Mathematical Tables and Other Aids to Computation*, vol. 6, p. 255. [2]


## 6.4 不完全B函数、学生分布、F分布、累积二项式分布

不完全B函数(beta函数)定义为

$$I_x(a,b) \equiv \frac{B_x(a,b)}{B(a,b)} \equiv \frac{1}{B(a,b)}\int_0^x t^{a-1}(1-t)^{b-1}dt \quad (a,b>0) \qquad (6.4.1)$$

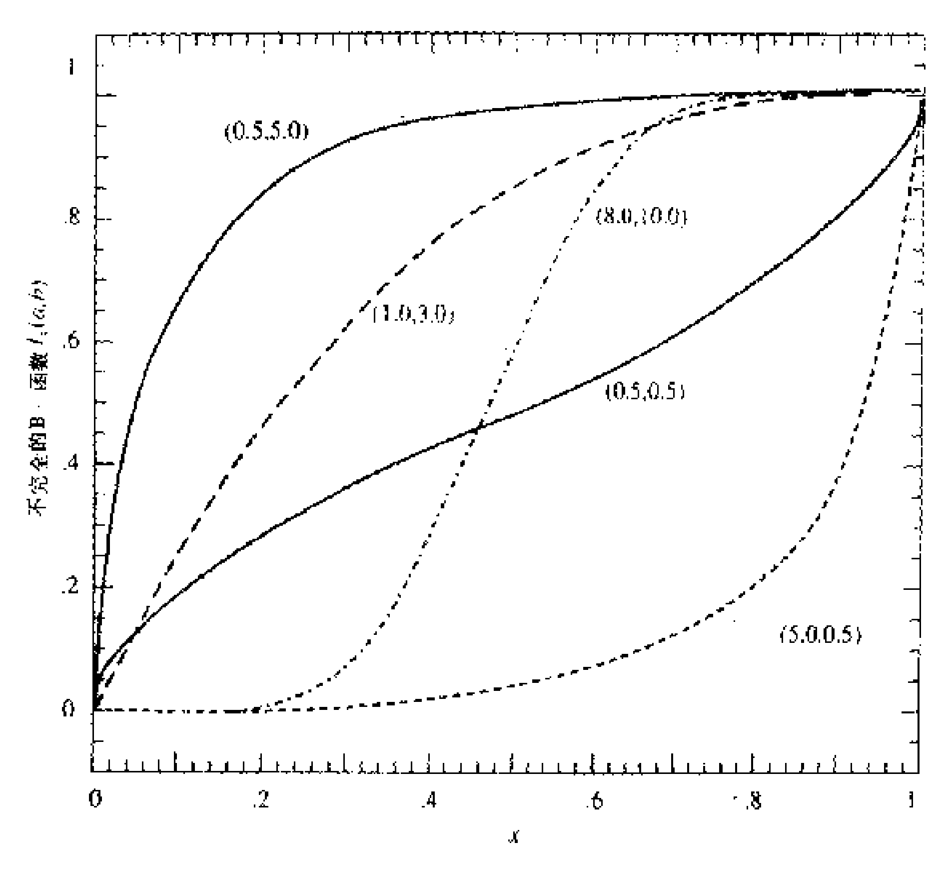
有限定值为

$$I_0(a,b)=0 \qquad I_1(a,b)=1 \tag{6.4.2}$$

及对称关系

$$I_x(a,b)=1-I_{1-x}(b,a) \tag{6.4.3}$$

若 $a$、$b$ 远大于1，则 $I_x(a,b)$ 在约 $x=a/(a+b)$ 处从接近零快速增长到接近1。图6.4.1画出了几对 $(a,b)$ 的函数曲线。




注意(0.5,5.0)和(5.0,0.5)关于对角线反射对称(参见式6.3.3)。

图6.4.1　五对不同 $(a,b)$ 的不完全B函数 $I_x(a,b)$

不完全B函数级数展开式为

$$I_x(a,b)=\frac{x^a(1-x)^b}{a\mathrm{B}(a,b)}\left[1+\sum_{n=0}^{\infty}\frac{\mathrm{B}(a+1,n+1)}{\mathrm{B}(a+b,n+1)}x^{n+1}\right] \tag{6.4.4}$$

此式没有证明，但它在数值计算过程中是很有用的。(不过，能够注意运用式(6.1.8)和(6.1.3)，对每个 $n$ 值由前面结果和一些乘法后，就可求得系数中的B函数值)。

连分式表示已证明是更为有用，

$$I_x(a,b)=\frac{x^a(1-x)^b}{aB(a,b)}\left[\frac{1}{1+}\frac{d_1}{1+}\frac{d_2}{1+}\cdots\right] \tag{6.4.5}$$

其中

$$d_{2m+1}=-\frac{(a+m)(a+b+m)x}{(a+2m)(a+2m+1)}$$

$$d_{2m} = \frac{m(b-m)x}{(a+2m-1)(a+2m)} \qquad (6.4.6)$$

连分式当 $x<(a+1)/(a+b+2)$时收敛很快，最坏的情况是 $O(\sqrt{max(a,b)})$次迭代。当 $x>(a+1)/(a+b-2)$时，可利用式(6.4.3)的对称关系获得等价的计算，那时连分式也快速收敛。因此有

```
#include <math.h>

float betai(float a, float b, float x)        返回不完全 B 函数值 Ix(a,b)
{
    float betacf(float a, float b, float x);
    float gammln(float xx);
    void nrerror(char error_text[]);
    float bt;

    if (x < 0.0 || x > 1.0) nrerror("Bad x in routine betai");
    if (x == 0.0 || x == 1.0) bt=0.0;
    else                                                          连分式前的因子
        bt=exp(gammln(a+b)-gammln(a)-gammln(b)+a*log(x)+b*log(1.0-x));
    if (x < (a-1.0)/(a+b+2.0))                                    直接使用连分式
        return bt*betacf(a,b,x)/a;
    else                                                          作对称变换以后使用连分式
        return 1.0-bt*betacf(b,a,1.0-x)/b;
}
```

下面程序用连分式求值

```
#include <math.h>
#define MAXIT 100
#define EPS 3.0e-7
#define FPMIN 1.0e-30

float betacf(float a, float b, float x)     用修正 lentz 法对不完全 B 函数的连分式求值，被 betai 调用
{
    void nrerror(char error_text[]);
    int m,m2;
    float aa,c,d,del,h,qab,qam,qap;

    qab=a+b;                                这些 q 用在(6.4.6)式系数的因子中
    qap=a+1.0;
    qam=a-1.0;
    c=1.0;                                  Lentz方法的第一步
    d=1.0-qab*x/qap;
    if (fabs(d) < FPMIN) d=FPMIN;
    d=1.0/d;
    h=d;
    for (m=1;m<=MAXIT;m++) {
        m2=2*m;
        aa=m*(b-m)*x/((qam+m2)*(a+m2));
        d=1.0+aa*d;                         递推式的一步(偶数步)
        if (fabs(d) < FPMIN) d=FPMIN;
        c=1.0+aa/c;
        if (fabs(c) < FPMIN) c=FPMIN;
        d=1.0/d;
        h *= d*c;
        aa = -(a+m)*(qab+m)*x/((a+m2)*(qap+m2));
        d=1.0+aa*d;                         递推下一步(奇数步)
        if (fabs(d) < FPMIN) d=FPMIN;
        c=1.0+aa/c;
```

```
        if (fabs(c) < FPMIN) c=FPMIN;
        d=1.0/d;
        del=d*c;
        h *= del;
        if (fabs(del-1.0) < EPS) break;    做完了吗
    }
    if (m > MAXIT) nrerror("a or b too big, or MAXIT too small in betacf");
    return h;
}
```

### 6.4.1 学生分布概率函数

学生分布用 $A(t|\nu)$ 表示,它在统计学中很有用,特别是用于检验两个观察分布是否有相同均值。$A(t|\nu)$ 为有 $\nu$ 重自由度的概率,若两均值相等,则某统计量 $t$(用于度量均值差)小于观测值。(详见第十四章)。如果 $A(t|\nu)>0.99$,则两均值相差很大,换言之,$1-A(t|\nu)$ 就很小,说明均值相等的假设不成立。

函数的数学定义为

$$A(t|\nu)=\frac{1}{\nu^{1/2}B(\frac{1}{2},\frac{\nu}{2})}\int_{-t}^{t}\left(1+\frac{x^2}{\nu}\right)^{-\frac{\nu+1}{2}}dx \tag{6.4.7}$$

限定值为

$$A(0|\nu)=0 \qquad A(\infty|\nu)=1 \tag{6.4.8}$$

$A(t|\nu)$ 与不完全 B 函数 $I_x(a,b)$ 关系为

$$A(t|\nu)=1-I_{\frac{\nu}{\nu+t^2}}\left(\frac{\nu}{2},\frac{1}{2}\right) \tag{6.4.9}$$

这样,可用式(6.4.9)及上述程序 **Betai** 来求函数值。

### 6.4.2 *F* 分布概率函数

此函数用于统计中检验两个观察样本是否有同样的方差。统计量 $F$ 定义为:第一个样本的观测离差与第二个样本的观测离差之比,它不难计算得到。(详见第十四章)。若第一个样本分布方差小于第二个样本分布方差,则与 $F$ 一样大小的概率,记为 $Q(F|\nu_1,\nu_2)$,其中 $\nu_1,\nu_2$ 分别为第一个和第二个样本的自由度数。换言之,$Q(F|\nu_1,\nu_2)$ 是表示假设"第一个方差小于第二个方差"被否决的显著性水平。其数值很小,表明被否决的可能性很大;而其反过来的含义就是,假设"第一个方差大于等于第二个方差"的置信度很大。

$Q(F|\nu_1,\nu_2)$ 有限定值

$$Q(0|\nu_1,\nu_2)=1 \qquad Q(\infty|\nu_1,\nu_2)=0 \tag{6.4.10}$$

与不完全 B 函数 $I_x(a,b)$ 关系($I_x(a,b)$ 由前面 **betai** 得到):

$$Q(F|\nu_1,\nu_2)=I_{\frac{\nu_2}{\nu_2+\nu_1F}}\left(\frac{\nu_2}{2},\frac{\nu_1}{2}\right) \tag{6.4.11}$$

### 6.4.3 累积二项式概率分布

假设每次试验中某一事件发生概率为 $p$,则 $n$ 次试验中它发生次数超过 $k$(或等于 $k$)的概率 $P$ 称为**累积二项式概率**,与不完全 $B$ 函数关系如下:

$$P = \sum_{j=k}^{n} \binom{n}{j} p^j (1-p)^{n-j} = I_p(k, n-k+1) \tag{6.4.12}$$

当 $n$ 大于12左右时，最好用 **betai** 求式(6.4.12)中的和，而不用计算二项式系数再直接求和。(当n小于12时，两种方法均可)。

## 6.5 整数阶贝塞尔函数

本节及下一节将给出计算各种整数阶贝塞尔函数的一些实用算法。第6.7节讨论分数阶情况。实际上计算分数阶情况的程序对整数阶情况也能很好运行。不过，本节计算整数阶的函数要更简单快速。唯一的不足是，受到以有理逼近为基础的精度的限制。双精度情况时最好用第6.7节中分数阶函数。

贝塞尔函数对任何实数 $\nu$，由级数表示定义为

$$J_\nu(x) = \left(\frac{1}{2}x\right)^\nu \sum_{k=0}^{\infty} \frac{\left(-\frac{1}{4}x^2\right)^k}{k!\,\Gamma(\nu+k+1)} \tag{6.5.1}$$

此级数对任何 $x$ 都收敛，但 $x \gg 1$ 时，计算上不是很有用的。

当 $\nu$ 不是整数时，贝塞尔函数由下式给出

$$Y_\nu(x) = \frac{J_\nu(x)\cos(\nu\pi) - J_{-\nu}(x)}{\sin(\nu\pi)} \tag{6.5.2}$$

$\nu$ 趋近整数 $n$ 时，上式右边趋近于正确限定值 $Y_n(x)$，但在实用时没有多大用处。

若变量 $x<\nu$，定性地看，上面两类贝塞尔函数都呈简单的幂关系，对 $0<x\ll\nu$ 的渐近形式为

$$\begin{aligned} J_\nu(x) &\sim \frac{1}{\Gamma(\nu+1)}\left(\frac{1}{2}x\right)^\nu \qquad \nu \geqslant 0 \\ Y_0(x) &\sim \frac{2}{\pi}\ln(x) \\ Y_\nu(x) &\sim -\frac{\Gamma(\nu)}{\pi}\left(\frac{1}{2}x\right)^{-\nu} \qquad \nu > 0 \end{aligned} \tag{6.5.3}$$

若 $x>\nu$，定性地看，上面两类贝塞尔函数象正弦或余弦波，其幅度按 $x^{-\frac{1}{2}}$ 衰减。$x\gg\nu$ 的渐近形式为

$$\begin{aligned} J_\nu(x) &\sim \sqrt{\frac{2}{\pi x}}\cos\left(x - \frac{1}{2}\nu\pi - \frac{1}{4}\pi\right) \\ Y_\nu(x) &\sim \sqrt{\frac{2}{\pi x}}\sin\left(x - \frac{1}{2}\nu\pi - \frac{1}{4}\pi\right) \end{aligned} \tag{6.5.4}$$

在 $x\sim\nu$ 的过渡区间内，贝塞尔函数的典型幅度按

$$\begin{aligned} J_\nu(\nu) &\sim \frac{2^{1/3}}{3^{2/3}\Gamma\left(\frac{2}{3}\right)}\frac{1}{\nu^{1/3}} \sim \frac{0.4473}{\nu^{1/3}} \\ Y_\nu(\nu) &\sim -\frac{2^{1/3}}{3^{1/6}\Gamma\left(\frac{2}{3}\right)}\frac{1}{\nu^{1/3}} \sim -\frac{0.7748}{\nu^{1/3}} \end{aligned} \tag{6.5.5}$$

$x$很大时，上式渐近式有效。图6.5.1画出了每类贝塞尔函数前$n$个图形。

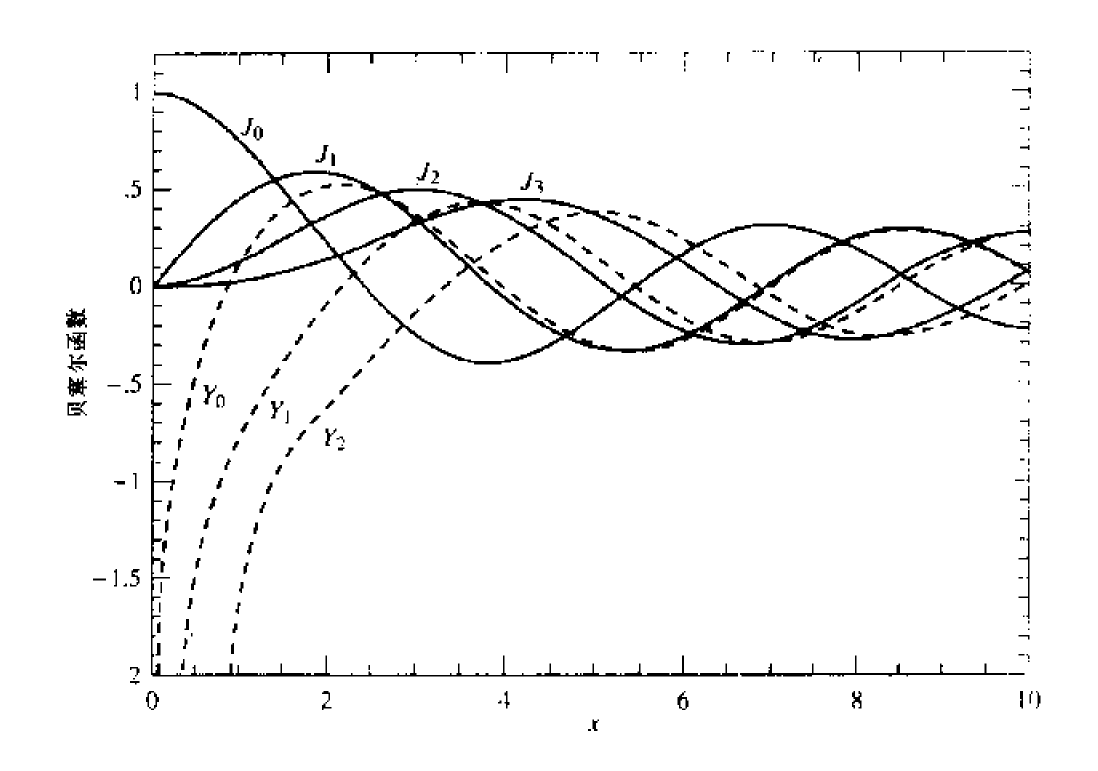



图6.5.1 $J_0(x)$到$J_3(x)$，$Y_0(x)$到$Y_2(x)$的贝塞尔函数

贝塞尔函数满足递推关系

$$J_{n+1}(x) = \frac{2n}{x}J_n(x) - J_{n-1}(x) \tag{6.5.6}$$

及

$$Y_{n+1}(x) = \frac{2n}{x}Y_n(x) - Y_{n-1}(x) \tag{6.5.7}$$

如第5.5节已指出，当$x<n$时，沿$n$增加的方向只有式(6.5.7)的第二部分是稳定的。沿$n$增加的方向式(6.5.6)是不稳定的，其原因是它与式(6.5.7)的递推关系一样：由舍入误差引入的少量"污染"了的$Y_n$很快就使求$J_n$遇到麻烦，见式(6.5.3)。

计算整数阶贝塞尔函数的实用策略分成两步：第一步，如何计算$J_0$、$J_1$及$Y_0$、$Y_1$；第二步，如何使用稳定递推关系找到其它$J$和$Y$。首先论述第一步的任务：

对落在0至任意值(取为8)之间的$x$，用$x$的有理函数逼近$J_0(x)$和$J_1(x)$。同样地，用有理函数去逼近$Y_0(x)$和$Y_1(x)$的"正则部分"，此部分定义为

$$Y_0(x) - \frac{2}{\pi}J_0(x)\ln(x) \quad 及 \quad Y_1(x) - \frac{2}{\pi}\left[J_1(x)\ln(x) - \frac{1}{x}\right] \tag{6.5.8}$$

当$8<x<\infty$时，用逼近式($n=0,1$)

$$J_n(x) = \sqrt{\frac{2}{\pi x}}\left[P_n\left(\frac{8}{x}\right)\cos(X_n) - Q_n\left(\frac{8}{x}\right)\sin(X_n)\right] \tag{6.5.9}$$

$$Y_n(x) = \sqrt{\frac{2}{\pi x}}\left[P_n\left(\frac{8}{x}\right)\sin(X_n) - Q_n\left(\frac{8}{x}\right)\cos(X_n)\right] \tag{6.5.10}$$

其中

$$X_n \equiv x - \frac{2n+1}{4}\pi \qquad (6.5.11)$$

$0<\frac{8}{x}<1$时，$P_0$，$P_1$，$Q_0$，$Q_1$是其变量的多项式。$P$ 为偶多项式，$Q$ 为奇多项式。

哈特一对不同的精度要求，给出了不同有理函数和多项式的系数。直接实现为

```
include <math.h>

float bessj0(float x)        对任意实数 x 返回贝塞尔函数 J0(x)

{
    float ax,z;
    double xx,y,ans,ans1,ans2;              按双精度累积多项式

    if ((ax=fabs(x)) < 8.0) {               直接有理函数逼近
        y=x*x;
        ans1=57568490574.0+y*(-13362590354.0+y*(651619640.7
            +y*(-11214424.18+y*(77392.33017+y*(-184.9052456)))));
        ans2=57568490411.0+y*(1029532985.0+y*(9494680.718
            +y*(59272.64853+y*(267.8532712+y*1.0))));
        ans=ans1/ans2;
    } else {                                拟合函数(6.5.9)式
        z=8.0/ax;
        y=z*z;
        xx=ax-0.785398164;
        ans1=1.0+y*(-0.1098628627e-2+y*(0.2734510407e-4
            +y*(-0.2073370639e-5+y*0.2093887211e-6)));
        ans2 = -0.1562499995e-1+y*(0.1430488765e-3
            +y*(-0.6911147651e-5+y*(0.7621095161e-6
            -y*0.934935152e-7)));
        ans=sqrt(0.636619772/ax)*(cos(xx)*ans1-z*sin(xx)*ans2);
    }
    return ans;
}
```

```
include <math.h>

float bessy0(float x)        对正数 x 返回贝塞尔函数 Y0(x)

{
    float bessj0(float x);
    float z;
    double xx,y,ans,ans1,ans2;              按双精度累积多项式

    if (x < 8.0) {                          (6.4.8)式有理函数逼近
        y=x*x;
        ans1 = -2957821389.0+y*(7062834065.0+y*(-512359803.6
            +y*(10879881.29+y*(-86327.92757+y*228.4622733))));
        ans2=40076544269.0+y*(745249964.8+y*(7189466.438
            +y*(47447.26470+y*(226.1030244+y*1.0))));
        ans=(ans1/ans2)+0.636619772*bessj0(x)*log(x);
    } else {                                拟合函数(6.5.10)式
        z=8.0/x;
        y=z*z;
        xx=x-0.785398164;
        ans1=1.0+y*(-0.1098628627e-2+y*(0.2734510407e-4
            +y*(-0.2073370639e-5+y*0.2093887211e-6)));
        ans2 = -0.1562499995e-1+y*(0.1430488765e-3
            +y*(-0.6911147651e-5+y*(0.7621095161e-6
            +y*(-0.934945152e-7))));
        ans=sqrt(0.636619772/x)*(sin(xx)*ans1+z*cos(xx)*ans2);
```

```
    }
    return ans;
}
#include <math.h>

float bessj1(float x)        对任意实数x返回贝塞尔函数值J1(x)
{
    float ax,z;
    double xx,y,ans,ans1,ans2;           按双精度累积多项式

    if ((ax=fabs(x)) < 8.0) {            直接有理逼近
        y=x*x;
        ans1=x*(72362614232.0+y*(-7895059235.0+y*(242396853.1
            +y*(-2972611.439+y*(15704.48260+y*(-30.16036606))))));
        ans2=144725228442.0+y*(2300535178.0+y*(18583304.74
            +y*(99447.43394+y*(376.9991397+y*1.0))));
        ans=ans1/ans2;
    } else {                             拟合函数(6.5.9)式
        z=8.0/ax;
        y=z*z;
        xx=ax-2.356194491;
        ans1=1.0+y*(0.183105e-2+y*(-0.3516396496e-4
            +y*(0.2457520174e-5+y*(-0.240337019e-6))));
        ans2=0.04687499995+y*(-0.2002690873e-3
            +y*(0.8449199096e-5+y*(-0.88228987e-6
            +y*0.105787412e-6)));
        ans=sqrt(0.636619772/ax)*(cos(xx)*ans1-z*sin(xx)*ans2);
        if (x < 0.0) ans = -ans;
    }
    return ans;
}
```

```
#include <math.h>

float bessy1(float x)        对正数x返回贝塞尔函数值Y1(x)
{
    float bessj1(float x);
    float z;
    double xx,y,ans,ans1,ans2;           按双精度累积多项式

    if (x < 8.0) {                       (6.5.8)式有理函数逼近
        y=x*x;
        ans1=x*(-0.4900604943e13+y*(0.1275274390e13
            +y*(-0.5153438139e11+y*(0.7349264551e9
            +y*(-0.4237922726e7+y*0.8511937935e4)))));
        ans2=0.2499580570e14+y*(0.4244419664e12
            +y*(0.3733650367e10+y*(0.2245904002e8
            +y*(0.1020426050e6+y*(0.3549632885e3+y)))));
        ans=(ans1/ans2)+0.636619772*(bessj1(x)*log(x)-1.0/x);
    } else {                             拟合函数(6.5.10)式
        z=8.0/x;
        y=z*z;
        xx=x-2.356194491;
        ans1=1.0+y*(0.183105e-2+y*(-0.3516396496e-4
            +y*(0.2457520174e-5+y*(-0.240337019e-6))));
        ans2=0.04687499995+y*(-0.2002690873e-3
            +y*(0.8449199096e-5+y*(-0.88228987e-6
            +y*0.105787412e-6)));
        ans=sqrt(0.636619772/x)*(sin(xx)*ans1+z*cos(xx)*ans2);
    }
```

```
    return ans;
}
```

现在讨论第二步，即如何由递推关系(6.5.6)和(6.5.7)得到贝塞尔函数 $J_n(x)$ 和 $Y_n(x)$ $(n\geqslant 2)$。$Y_n(x)$ 可直接算，因其向前递推总是稳定的。

```
float bessy(int n,float x)       对正数 x 及 n≥2返回贝塞尔函数值 Yn(x)
{
    float bessy0(float x);
    float bessy1(float x);
    void nrerror(char error_text[]);
    int j;
    float by,bym,byp,tox;

    if (n < 2) nrerror("Index n less than 2 in bessy");
    tox=2.0/x;
    by=bessy1(x);                        递推式的起始值
    bym=bessy0(x);
    for (j=1;j<n;j++) {                  递推式(6.5.7)
        byp=j * tox * by-bym;
        bym=by;
        by=byp;
    }
    return by;
}
```

本算法的代价是调用 **bessy1** 和 **bessy0**(导致调用各自的 **bessj1** 和 **bessj0**)，再加上递推中 $O(n)$ 次运算。

至于 $J_n(x)$，就有点复杂了。可以从 $J_0$ 和 $J_1$ 开始，按 $n$ 向上递推，但仅当 $n$ 不超过 $x$ 时，递推保持稳定。这样仅当 $x$ 很大而 $n$ 较小时才适合，不过实际经常正是如此。

困难的是当 $x<n$，这时最好用米勒(Miller)算法(见前式(5.5.16)的讨论)，从任意起始值开始向后递推，利用递推式的向前不稳定性而到达正确解。最后得到 $J_0$ 和 $J_1$ 时，再用过程中的累积和式(5.5.16)规一化所得到的解。

唯一的问题是，应从多大的 $n$ 处开始向后递推，使达到希望的 $n$ 值时能得到要求的精度。考察渐近式(6.5.3)和(6.5.5)，可以得出结论，应从比期望的 $n$ 值大一个增量$[常数\times n]^{1/2}$ 的地方开始，其中常数的平方根粗略地等于精度的有效位位数。

由上面讨论得出下面程序函数。

```
#include <math.h>
#define ACC 40.0
#define BIGNO 1.0e10                 加大以提高精度
#define BIGNI 1.0e-10

float bessj(int n, float x)      对任意实数 x 及 n≥2返回贝塞尔函数值 Jn(x)
{
    float bessj0(float x);
    float bessj1(float x);
    void nrerror(char error_text[]);
    int j,jsum,m;
    float ax,bj,bjm,bjp,sum,tox,ans;
```

```
    if (n < 2) nrerror("Index n less than 2 in bessj");
    ax=fabs(x);
    if (ax == 0.0)
        return 0.0;
    else if (ax > (float) n) {          从J0,J1开始向前递推
        tox=2.0/ax;
        bjm=bessj0(ax);
        bj=bessj1(ax);
        for (j=1;j<n;j++) {
            bjp=j*tox*bj-bjm;
            bjm=bj;
            bj=bjp;
        }
        ans=bj;
    } else {                            从这儿算得的偶数m开始向后递推
        tox=2.0/ax;
        m=2*((n+(int) sqrt(ACC*n))/2);
        jsum=0;                         jsum取得0或1,当取1时,求(5.5.16)式中偶数
        bjp=ans=sum=0.0;                    项之和
        bj=1.0;
        for (j=m;j>0;j--) {             向后递推
            bjm=j*tox*bj-bjp;
            bjp=bj;
            bj=bjm;
            if (fabs(bj) > BIGNO) {     重新归一化以防溢出
                bj *= BIGNI;
                bjp *= BIGNI;
                ans *= BIGNI;
                sum *= BIGNI;
            }
            if (jsum) sum += bj;        求累积和
            jsum=!jsum;                 把0变为1或把1变为0
            if (j == n) ans=bjp;        存储未规一化的结果
        }
        sum=2.0*sum-bj;                 计算(5.5.16)
        ans /= sum;                     用于答案的归一化
    }
    return x < 0.0 && (n & 1) ? -ans : ans;
}
```

参考文献和进一步读物:


Hart, J.F., et al. 1968, *Computer Approximations* (New York: Wiley), §6.8, p. 141. [1]


## 6.6 修正的整数阶贝塞尔函数

修正贝塞尔函数 $I_n(x)$ 和 $K_n(x)$ 对纯虚数变量与一般贝塞尔函数 $J_n$ 和 $Y_n$ 等价。详细地说,其关系式为:

$$I_n(x) = (-i)^n J_n(ix)$$

$$K_n(x) = \frac{\pi}{2} i^{n+1} \left[ J_n(ix) + iY_n(ix) \right] \tag{6.6.1}$$

仔细选择特殊前因子和组成 $K_n$ 的 $J_n$ 和 $Y_n$ 的线性组合,使得变量 $x$ 为实数时,函数值为实数。

$x$ 很小($x \ll n$)时 $I_n(x)$ 和 $K_n(x)$ 都逐渐逼近其自变量的简单幂函数

$$I_n(x) \approx \frac{1}{n!}\left(\frac{x}{2}\right)^n \qquad n \geqslant 0$$

$$K_0(x) \approx -\ln(x)$$

$$K_n(x) \approx \frac{(n-1)!}{2}\left(\frac{x}{2}\right)^{-n} \qquad n > 0 \tag{6.6.2}$$

上面表达式实质上与 $J_n(x)$、$Y_n(x)$ 在此区域内的渐近式相同，只是 $Y_n(x)$ 和 $K_n(x)$ 相差一个因子 $-\frac{2}{\pi}$。

$x \gg n$ 区域内，修正贝塞尔函数具有与贝塞尔函数完全不同的性质，

$$\begin{aligned} I_n(x) &\approx \frac{1}{\sqrt{2\pi x}} \exp(x) \\ K_n(x) &\approx \frac{\pi}{\sqrt{2\pi x}} \exp(-x) \end{aligned} \tag{6.6.3}$$

自变量很大时，修正函数明显具有指数而不是正弦性质(见图6.6.1)。

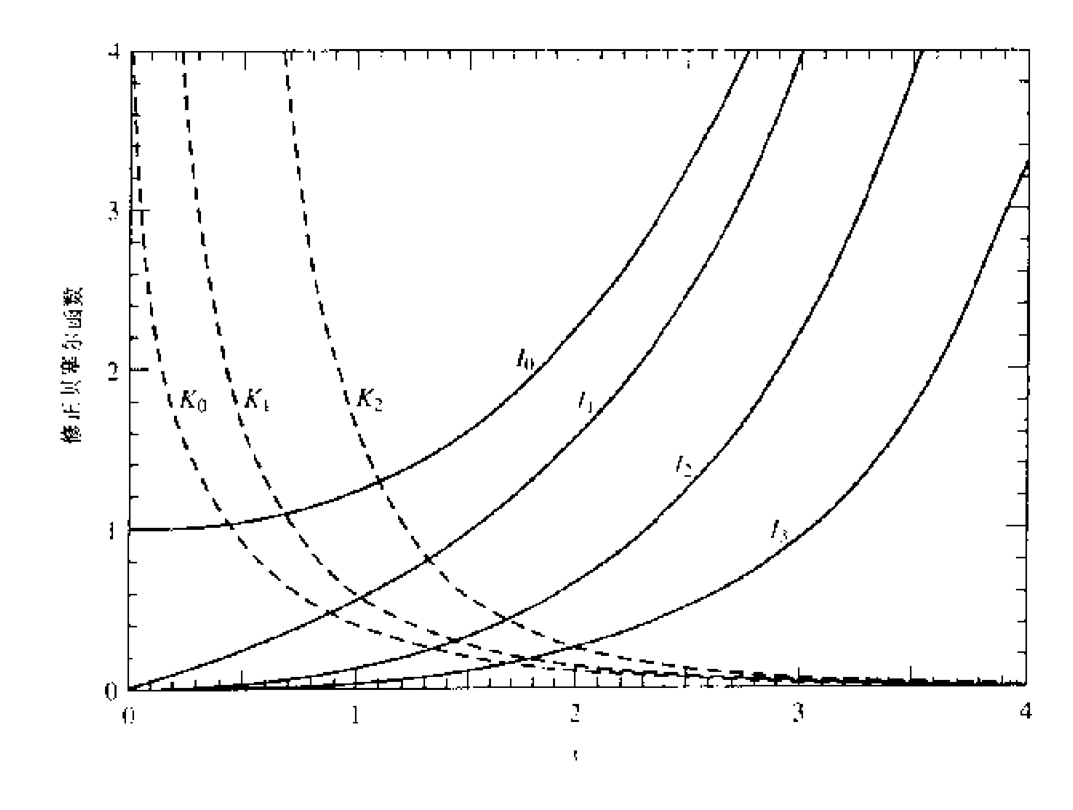



图6.6.1 修正贝塞尔函数 $I_0(x)$、$I_1(x)$、$I_2(x)$、$K_0(x)$、$K_1(x)$、$K_2(x)$

若去掉指数因子则修正贝塞尔函数较平滑，这一点使只含有 $n$ 项的简单多项式的逼近，很适用于 $I_0$、$I_1$、$K_0$ 及 $K_1$。下面程序基于由 Abramowitz 和 Stegun[1]给出的多项式系数，计算了此四个函数值，同时可用于 $x > n$ 时，对 $n > 1$ 的向前递推。

```
#include <math.h>

float bessi0(float x)          对任意实数 x 返回修正贝塞尔函数值 I0(x)
{
    float ax,ans;
    double y;                                  按双精度累积多项式
```

```
    if ((ax=fabs(x)) < 3.75) {          多项式拟合
        y=x/3.75;
        y*=y;
        ans=1.0+y*(3.5156229+y*(3.0899424+y*(1.2067492
            +y*(0.2659732+y*(0.360768e-1+y*0.45813e-2)))));
    } else {
        y=3.75/ax;
        ans=(exp(ax)/sqrt(ax))*(0.39894228+y*(0.1328592e-1
            +y*(0.225319e-2+y*(-0.157565e-2+y*(0.916281e-2
            +y*(-0.2057706e-1+y*(0.2635537e-1+y*(-0.1647633e-1
            +y*0.392377e-2))))))));
    }
    return ans;
}
```

```
#include <math.h>

float bessk0(float x)       对正实数 x 返回修正贝塞尔函数值 K_0(x)
{
    float bessi0(float x);
    double y,ans;                   按双精度累积多项式
    if (x <= 2.0) {                 多项式拟合
        y=x*x/4.0;
        ans=(-log(x/2.0)*bessi0(x))+(-0.57721566+y*(0.42278420
            +y*(0.23069756+y*(0.3488590e-1+y*(0.262698e-2
            +y*(0.10750e-3+y*0.74e-5))))));
    } else {
        y=2.0/x;
        ans=(exp(-x)/sqrt(x))*(1.25331414+y*(-0.7832358e-1
            +y*(0.2189568e-1+y*(-0.1062446e-1+y*(0.587872e-2
            +y*(-0.251540e-2+y*0.53208e-3))))));
    }
    return ans;
}
```

```
#include <math.h>

float bessi1(float x)       对任意实数 x 返回修正贝塞尔函数值 I_1(x)
{
    float ax,ans;
    double y;                               按双精度累积多项式

    if ((ax=fabs(x)) < 3.75) {              多项式拟合
        y=x/3.75;
        y*=y;
        ans=ax*(0.5+y*(0.87890594+y*(0.51498869+y*(0.15084934
            +y*(0.2658733e-1+y*(0.301532e-2+y*0.32411e-3))))));
    } else {
        y=3.75/ax;
        ans=0.2282967e-1+y*(-0.2895312e-1+y*(0.1787654e-1
            -y*0.420059e-2));
        ans=0.39894228+y*(-0.3988024e-1+y*(-0.362018e-2
            +y*(0.163801e-2+y*(-0.1031555e-1+y*ans))));
        ans *= (exp(ax)/sqrt(ax));
    }
    return x < 0.0 ? -ans : ans;
}
```

```
#include <math.h>
```

```
float bessk1(float x)       对正实数 x 返回修正贝塞尔函数值 K1(x)
{
    float bessi1(float x);
    double y,ans;                   按双精度累积多项式

    if (x <= 2.0) {                 多项式拟合
        y=x*x/4.0;
        ans=(log(x/2.0)*bessi1(x))+(1.0/x)*(1.0+y*(0.15443144
            +y*(-0.67278579+y*(-0.18156897+y*(-0.1919402e-1
            +y*(-0.110404e-2+y*(-0.4686e-4)))))));
    } else {
        y=2.0/x;
        ans=(exp(-x)/sqrt(x))*(1.25331414+y*(0.23498619
            +y*(-0.3655620e-1+y*(0.1504268e-1+y*(-0.780353e-2
            +y*(0.325614e-2+y*(-0.68245e-3)))))));
    }
    return ans;
}
```

$I_n(x)$，$K_n(x)$的递推关系与 $J_n(x)$，$Y_n(x)$相同，只是用 $ix$ 代替 $x$，这样在关系式中就需改变符号，

$$
\begin{aligned}
I_{n-1}(x) &= -\left(\frac{2n}{x}\right)I_n(x) + I_{n+1}(x) \\
K_{n+1}(x) &= +\left(\frac{2n}{x}\right)K_n(x) + K_{n-1}(x)
\end{aligned}
\tag{6.6.4}
$$

向前递推时，这些关系式常常是不稳定的。但对 $K_n$，本身增长没有问题。对 $I_n$，需再次用向后递推的方法，并按 bessj 程序相同的方法选择递推起始点。根本的区别就在于，$I_n(x)$ 的规一化公式中连续项交替出现减号，这也是因为在先前对 $J_n$ 使用的公式中，用 $ix$ 代替 $x$ 的缘故。

$$1 = I_0(x) - 2I_2(x) + 2I_4(x) - 2I_6(x) + \cdots \tag{6.6.5}$$

实际上，简单地调用 bessi0进行规一化更好。

**递推程序 bessj 和 bessy 经过简单修改后就成为新程序 bessi 和 bessk：**

```
float bessk(int n, float x)     对正数 x 及 n≥2返回修正贝塞尔函数值 Kn(x)
{
    float bessk0(float x);
    float bessk1(float x);
    void nrerror(char error_text[]);
    int j;
    float bk,bkm,bkp,tox;

    if (n < 2) nrerror("Index n less than 2 in bessk");
    tox=2.0/x;
    bkm=bessk0(x);                      对所有 x 向前递推
    for (j=1;j<n;j++) {                 这里做递推
        bkp=bkm+j*tox*bk;
        bkm=bk;
        bk=bkp;
    }
    return bk;
}

#include <math.h>
#define ACC 40.0                        增大以提高精度
#define BIGNO 1.0e10
```

```
#define BIGNI 1.0e-10

float bess(int n, float x)        对任意实数x及n≥2返回修正贝塞尔函数值 I_n(x)
{
    float bessi0(float x);
    void nrerror(char error_text[]);
    int j;
    float bi,bim,bip,tox,ans;

    if (n < 2) nrerror("Index n less than 2 in bessi");
    if (x == 0.0)
        return 0.0;
    else {
        tox=2.0/fabs(x);
        bip=ans=0.0;
        bi=1.0;
        for (j=2*(n+(int) sqrt(ACC*n));j>0;j--) {     从一个偶数m起向后递推
            bim=bip+j*tox*bi;
            bip=bi;
            bi=bim;
            if (fabs(bi) > BIGNO) {        重新规一化以防溢出
                ans *= BIGNI;
                bi *= BIGNI;
                bip *= BIGNI;
            }
            if (j == n) ans=bip;
        }
        ans *= bessi0(x)/bi;              用bessi0 进行规一化
        return x < 0.0 && (n & 1) ? -ans : ans;
    }
}
```

参考文献和进一步读物：


Abramowitz, M., and Stegun, I. A 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, vol. 55, §9.8. [1]

Carrier, G. F., Krook, M. and Pearson, C. E. 1966, *Functions of a Complex Variable*(New York: McGraw-Hill), pp. 220ff.


## 6.7 分数阶贝塞尔函数、Airy 函数、球面贝塞尔函数

数值地计算分数阶贝塞尔函数有很多方法。只是大多数算法并不实用。本节给出的程序虽然比较复杂，但确值得推崇。

### 6.7.1 一般贝塞尔函数

基本思想是**斯蒂特(Steed)法**，此方法来源于求 Coulomb 波形函数[1]。这方法必须同时求 $J_\nu$、$J'_\nu$、$Y_\nu$ 和 $Y'_\nu$，这样必然涉及四者之间的关系。有三个关系式来自两个连分式，其中有一式是复数。第四个关系式满足 Wronskian 关系：

$$W \equiv J_\nu Y'_\nu - Y_\nu J'_\nu = \frac{2}{\pi x} \tag{6.7.1}$$

第一个连分式 CF1定义为

$$f_\nu \equiv \frac{J'_\nu}{J_\nu} = \frac{\nu}{x} - \frac{J_{\nu+1}}{J_\nu} = \frac{\nu}{x} - \frac{1}{2(\nu+1)/x -}\ \frac{1}{2(\nu+2)/x -}\cdots \tag{6.7.2}$$

此式很容易由贝塞尔函数的三项递推式导出：从式(6.5.6)开始并利用式(5.5.18)，第5.2节连分式的前向

求值，本质上与递推关系式的后向递推等价。CF1的收敛速率由**转折点** $x_{tp}=\sqrt{\nu(\nu-1)}\approx\nu$ 的位置决定，在此以外贝塞尔函数为振荡性。$x\lesssim x_{tp}$时，收敛很快。若 $x\gtrsim x_{tp}$，每次 $\nu$ 增加1，迭代连分式直到 $x\lesssim x_{tp}$，此时收敛加快。因此 $x$ 很大时，CF1的迭代次数与 $x$ 同阶。程序 **bessjy** 中，设定允许迭代次数为 10000，$x$ 很大时，则使用贝塞尔函数的一般渐近表达式。

可以看出一旦CF1收敛，它的分母便与 $J_\nu$ 的符号一致。

复连分式CF2定义为

$$p+iq\equiv\frac{J'_\nu+iY'_\nu}{J_\nu+iY_\nu}=-\frac{1}{2x}+i+\frac{i}{x}\,\frac{(1/2)^2-\nu^2}{2(x+i)+}\,\frac{(3/2)^2-\nu^2}{2(x+2i)+}\cdots \tag{6.7.3}$$

（在下一小节中，讨论如何按类似修正贝塞尔函数方式导出CF2。）$x\gtrsim x_{tp}$时，连分式收敛很快，$x\to0$时不收敛。$x$ 很小时，必须采用一种特殊方法，下面将要介绍。$x$ 不是很小时，保证 $x\gtrsim x_{tp}$，则将 $J_\nu$ 和 $J'_\nu$ 稳定地向后递推到 $\nu=\mu\lesssim x$，在比较小 $\nu$ 值处得到比值 $f_\mu$。递推关系式在此方向上稳定的。递推初始值是：

$$J_\nu=\text{任意},\qquad J'_\nu=f_\nu J_\nu \tag{6.7.4}$$

注意，$J_\nu$ 任意初始值的符号与CF1数的分母的符号相同。$J_\nu$ 的初始值应很小，以免在递推时产生溢出的可能。递推式为：

$$\begin{aligned}J_{\nu-1}&=\frac{\nu}{x}J_\nu+J'_\nu\\ J'_{\nu-1}&=\frac{\nu-1}{x}J_{\nu-1}-J_\nu\end{aligned} \tag{6.7.5}$$

一旦求得CF2在 $\nu=\mu$ 处值，则利用 Wronskin 式(6.7.1)，我们就有足够的关系式来求解四个量，引入一个变量可简化公式：

$$\gamma\equiv\frac{p-f_\mu}{q} \tag{6.7.6}$$

有

$$J_\mu=\pm\left(\frac{W}{q+\gamma(p-f_\mu)}\right)^{1/2} \tag{6.7.7}$$

$$J'_\mu=f_\mu J_\mu \tag{6.7.8}$$

$$Y_\mu=\gamma J_\mu \tag{6.7.9}$$

$$Y'_\mu=Y_\mu\left(p+\frac{q}{\gamma}\right) \tag{6.7.10}$$

式(6.7.7)中 $J_\mu$ 的符号应与式(6.7.4)中的 $J_\nu$ 的初始值的符号相同。

一旦得到 $\nu=\mu$ 处四个函数，就可以求得 $\nu$ 初始值处的函数值，对 $J_\nu$ 和 $J'_\nu$，只需用式(6.7.7)与使用式(6.7.5)递推后得到的值，代入式(6.7.4)就可简单地求得。又由式(6.7.9)和(6.7.10)中的值开始并使用稳定向前递推式，则可得 $Y_\nu$ 和 $Y'_\nu$

$$Y_{\nu+1}=\frac{2\nu}{x}Y_\nu-Y_{\nu-1} \tag{6.7.11}$$

同时应用关系式

$$Y'_\nu=\frac{\nu}{x}Y_\nu-Y_{\nu+1} \tag{6.7.12}$$

现在讨论 $x$ 较小时CF2不合适的情况。Temme[7]找到一种很好的方法，即通过精确处理 $x\to0$时奇异点的级数展开，来求 $Y_\nu$ 和 $Y_{\nu+1}$，然后由式(6.7.12)得到 $Y'_\nu$。展开式仅对 $|\nu|\le1/2$时有效，故在此范围内采用递推式(6.7.5)，求 $f_\nu$ 在 $\nu=\mu$ 处的值。接着由下式求 $J_\mu$

$$J_\mu=\frac{W}{Y'_\mu-Y_\mu f_\mu} \tag{6.7.13}$$

由式(6.7.8)，得到 $J'_\mu$。在 $\nu$ 初始值处的值 $J_\nu$ 和 $J'_\nu$ 可用如前所述方法求得，$Y$ 也如前所述可递推求得。

Temme 级数为

$$Y_\nu = -\sum_{k=0}^{\infty} c_k g_k \qquad Y_{\nu+1} = -\frac{2}{x}\sum_{k=0}^{\infty} c_k h_k \tag{6.7.14}$$

其中

$$c_k = \frac{(-x^2/4)^k}{k!} \tag{6.7.15}$$

系数 $g_k$ 和 $h_k$ 由 $p_k$、$q_k$ 和 $f_k$ 按递推式定义：

$$\begin{aligned} g_k &= f_k + \frac{2}{\nu}\sin^2\left(\frac{\nu\pi}{2}\right) q_k \\ h_k &= -k g_k + p_k \\ p_k &= \frac{p_{k-1}}{k-\nu} \\ q_k &= \frac{q_{k-1}}{k+\nu} \\ f_k &= \frac{k f_{k-1} + p_{k-1} + q_{k-1}}{k^2 - \nu^2} \end{aligned} \tag{6.7.16}$$

递推起始值为：

$$\begin{aligned} p_0 &= \frac{1}{\pi}\left(\frac{x}{2}\right)^{-\nu}\Gamma(1+\nu) \\ q_0 &= \frac{1}{\pi}\left(\frac{x}{2}\right)^{\nu}\Gamma(1-\nu) \\ f_0 &= \frac{2}{\pi}\,\frac{\nu\pi}{\sin\nu\pi}\left[\cosh\sigma\,\Gamma_1(\nu) + \frac{\sinh\sigma}{\sigma}\ln\left(\frac{2}{x}\right)\Gamma_2(\nu)\right] \end{aligned} \tag{6.7.17}$$

其中

$$\begin{aligned} \sigma &= \nu\ln\left(\frac{2}{x}\right) \\ \Gamma_1(\nu) &= \frac{1}{2\nu}\left[\frac{1}{\Gamma(1-\nu)} - \frac{1}{\Gamma(1+\nu)}\right] \\ \Gamma_2(\nu) &= \frac{1}{2}\left[\frac{1}{\Gamma(1-\nu)} + \frac{1}{\Gamma(1+\nu)}\right] \end{aligned} \tag{6.7.18}$$

公式这种写法的优越性是，通过小心计算 $\nu\pi/\sin\nu\pi$，$\sin\sigma/\sigma$ 及 $\Gamma_1$ 的值，能控制 $\nu\to 0$ 时可能出现的问题。特别是，Temme 给出了 $\Gamma_1(\nu)$ 和 $\Gamma_2(\nu)$ 的切比雪夫展开式。我们在这里将 $\Gamma_1$ 的展开式重新排成明晰的 $\nu$ 的偶级数形式，以便可以调用第5.8节中的程序 **Chebev**。

程序中假定 $\nu\geqslant 0$。若 $\nu$ 为负数，则可使用反射公式

$$\begin{aligned} J_{-\nu} &= \cos\nu\pi J_\nu - \sin\nu\pi Y_\nu \\ Y_{-\nu} &= \sin\nu\pi J_\nu + \cos\nu\pi Y_\nu \end{aligned} \tag{6.7.19}$$

程序还假定 $x>0$。$x<0$ 时一般函数很复杂，但可由 $x>0$ 时的函数来表示。$x=0$ 时，$Y_\nu$ 呈奇异性。

程序中算法使用双精度，复数运算通过实变量计算。

```
#include <math.h>
#include "nrutil.h"
#define EPS 1.0e-10
#define FPMIN 1.0e-30
#define MAXIT 10000
#define XMIN 2.0
#define PI 3.141592653589793

void bessjy(float x, float xnu, float *rj, float *ry, float *rjp, float *ryp)
    对正数 x 和 xnu=ν≥0返回贝塞尔函数 rj=J_ν, ry=Y_ν 及其导数 rjp=J'_ν, ryp=Y'_ν。除在某一函数零点附近处，相对
    精度在 EPS(EPS 是控制绝对精度)的一或两位有效数字范围之内。FPMIN 是接近机器最小浮点数的某一数。内部
```

算法均为双精度。欲将整个程序设置为双精度，只需将上现的变量浮点类型说明改为双精度类型说明.EPS 改为 $10^{-16}$ 同时修改函数 **beschb**

```
{
    void beschb(double x, double *gam1, double *gam2, double *gampl,
        double *gammi);
    int i,isign,l,nl;
    double a,b,br,bi,c,cr,ci,d,del,del1,den,di,dlr,dli,dr,e,f,fact,fact2,
        fact3,ff,gam,gam1,gam2,gammi,gampl,h,p,pimu,pimu2,q,r,rjl,
        rjl1,rjmu,rjp1,rjpl,rjtemp,ry1,rymu,rymup,rytemp,sum,sum1,
        temp,w,x2,xi,xi2,xmu,xmu2;

    if (x <= 0.0 || xnu < 0.0) nrerror("bad arguments in bessjy");
    nl=(x < XMIN ? (int)(xnu+0.5) : IMAX(0,(int)(xnu-x+1.5)));
    nl 是 J 的向后递推数及 Y 的向前递推数。当x<XMIN，xmu处于-1/2至1/2之间；
    而当x≥XMIN，xmu可选择，以使 x 大于转折点

    xmu=xnu-nl;
    xmu2=xmu*xmu;
    xi=1.0/x;
    xi2=2.0*xi;
    w=xi2/PI;                          Wronskian关系式
    isign=1;                           由修正Lentz法(§5.2)求CF1值，
    h=xnu*xi;                              isign指示分母符号的变化
    if (h < FPMIN) h=FPMIN;
    b=xi2*xnu;
    d=0.0;
    c=h;
    for (i=1;i<=MAXIT;i++) {
        b += xi2;
        d=b-d;
        if (fabs(d) < FPMIN) d=FPMIN;
        c=b-1.0/c;
        if (fabs(c) < FPMIN) c=FPMIN;
        d=1.0/d;
        del=c*d;
        h=del*h;
        if (d < 0.0) isign = -isign;
        if (fabs(del-1.0) < EPS) break;
    }
    if (i > MAXIT) nrerror("x too large in bessjy; try asymptotic expansion");
    rjl=isign*FPMIN;                   初始化 Jν 和 J'ν 准备向后递推
    rjpl=h*rjl;
    rjl1=rjl;                          存储值以便以后调整
    rjp1=rjpl;
    fact=xnu*xi;
    for (l=nl;l>=1;l--) {
        rjtemp=fact*rjl+rjpl;
        fact -= xi;
        rjpl=fact*rjtemp-rjl;
        rjl=rjtemp;
    }
    if (rjl == 0.0) rjl=EPS;
    f=rjpl/rjl;                        得到未规一化的 Jμ 和 J'μ
    if (x < XMIN) {                    使用级数
        x2=0.5*x;
        pimu=PI*xmu;
        fact = (fabs(pimu) < EPS ? 1.0 : pimu/sin(pimu));
        d = -log(x2);
        e=xmu*d;
        fact2 = (fabs(e) < EPS ? 1.0 : sinh(e)/e);
        beschb(xmu,&gam1,&gam2,&gampl,&gammi);    Γ1,Γ2的切比雪夫计算值
        ff=2.0/PI*fact*(gam1*cosh(e)+gam2*fact2*d);     f0.
        e=exp(e);
        p=e/(gampl*PI);                p0.
        q=1.0/(e*PI*gammi);            q0.
```

```
        pimu2=0.5*pimu;
        fact3 = (fabs(pimu2) < EPS ? 1.0 : sin(pimu2)/pimu2);
        r=PI*pimu2*fact3*fact3;
        c=1.0;
        d = -x2*x2;
        sum=ff+r*q;
        sum1=p;
        for (i=1;i<=MAXIT;i++) {
            ff=(i*ff+p+q)/(i*i-xmu2);
            c *= (d/i);
            p /= (i-xmu);
            q /= (i+xmu);
            del=c*(ff+r*q);
            sum += del;
            del1=c*p-i*del;
            sum1 += del1;
            if (fabs(del) < (1.0+fabs(sum))*EPS) break;
        }
        if (i > MAXIT) nrerror("bessy series failed to converge");
        rymu = -sum;
        ry1 = -sum1*xi2;
        rymup=xmu*xi*rymu-ry1;
        rjmu=w/(rymup-f*rymu);              方程(6.7.13)式
    } else {                                由修正Lentz法(求§5.2)求CF2值
        a=0.25-xmu2;
        p = -0.5*xi;
        q=1.0;
        br=2.0*x;
        bi=2.0;
        fact=a*xi/(p*p+q*q);
        cr=br+q*fact;
        ci=bi+p*fact;
        den=br*br+bi*bi;
        dr=br/den;
        di = -bi/den;
        dlr=cr*dr-ci*di;
        dli=cr*di+ci*dr;
        temp=p*dlr-q*dli;
        q=p*dli+q*dlr;
        p=temp;
        for (i=2;i<=MAXIT;i++) {
            a += 2*(i-1);
            bi += 2.0;
            dr=a*dr+br;
            di=a*di+bi;
            if (fabs(dr)+fabs(di) < FPMIN) dr=FPMIN;
            fact=a/(cr*cr+ci*ci);
            cr=br+cr*fact;
            ci=bi-ci*fact;
            if (fabs(cr)+fabs(ci) < FPMIN) cr=FPMIN;
            den=dr*dr+di*di;
            dr /= den;
            di /= -den;
            dlr=cr*dr-ci*di;
            dli=cr*di+ci*dr;
            temp=p*dlr-q*dli;
            q=p*dli+q*dlr;
            p=temp;
            if (fabs(dlr-1.0)+fabs(dli) < EPS) break;
        }
        if (i > MAXIT) nrerror("cf2 failed in bessjy");
        gam=(p-f)/q;                        方程(6.7.6)~(6.7.10)式
        rjmu=sqrt(w/((p-f)*gam+q));
        rjmu=SIGN(rjmu,rjl);
        rymu=rjmu*gam;
        rymup=rymu*(p+q/gam);
        ry1=xmu*xi*rymu-rymup;
```

```
    }
    fact=rjmu/rjl;
    *rj=rjl1*fact;                                 调整Jν和J'ν的起始值
    *rjp=rjp1*fact;
    for (i=1;i<=nl;i++) {                          Yν的向前递推
        rytemp=(xmu+i)*xi2*ry1-rymu;
        rymu=ry1;
        ry1=rytemp;
    }
    *ry=rymu;
    *ryp=xmu*xi*rymu-ry1;
}

#define NUSE1  5
#define NUSE2  5

void beschb(double x, double *gam1, double *gam2, double *gampl, double *gammi)
```

对于$|x|\leq 1/2$用切比雪夫展开式计算$\Gamma_1$和$\Gamma_2$。也返回$1/\Gamma(1+x)$和$1/\Gamma(1-x)$。若收敛到双精度，则设置$NUSE1=7$，$NUSE2=8$。

```
{
    float chebev(float a, float b, float c[], int m, float x);
    float xx;
    static float c1[] = {
        -1.142022680371172e0,6.516511267076e-3,
        3.087090173086e-4,-3.470626964e-6,6.943764e-9,
        3.6780e-11,-1.36e-13};
    static float c2[] = {
        1.843740587300906e0,-0.076852840844786e0,
        1.271927136655e-3,-4.971736704e-6,-3.3126120e-8,
        2.42310e-10,-1.70e-13,-1.0e-15};

    xx=8.0*x*x-1.0;                          x乘以2改变范围为-1到1
    *gam1=chebev(-1.0,1.0,c1,NUSE1,xx);        然后使用变换求偶切比雪夫级数值
    *gam2=chebev(-1.0,1.0,c2,NUSE2,xx);
    *gampl= *gam2-x*(*gam1);
    *gammi= *gam2+x*(*gam1);
}
```

## 6.7.2 修正贝塞尔函数

Steed 方法不能用于修正贝塞尔函数，因为此时CF2为纯虚数，四个函数中只有三个关系式。Temme[3]给出一个正则条件得到第四个关系式。

Wronskian 关系式为

$$W \equiv I_\nu K'_\nu - K_\nu I'_\nu = -\frac{1}{x} \tag{6.7.20}$$

连分式CF1变成为

$$f_\nu \equiv \frac{I'_\nu}{I_\nu} = \frac{\nu}{x} + \frac{1}{2(\nu+1)/x +} \frac{1}{2(\nu+2)/x +} \cdots \tag{6.7.21}$$

为方便地得到CF2及正则条件，考察合流超几何函数序列：

$$z_n(x) = U(\nu + 1/2 + n, 2\nu + 1, 2x) \tag{6.7.22}$$

$\nu$固定，则

$$K_\nu(x) = \pi^{1/2}(2x)^\nu e^{-x} z_0(x) \tag{6.7.23}$$

$$\frac{K_{\nu+1}(x)}{K_\nu(x)} = \frac{1}{x}\left[\nu + \frac{1}{2} + x + \left(\nu^2 - \frac{1}{4}\right)\frac{z_1}{z_0}\right] \tag{6.7.24}$$

式(6.7.23)是合流超几何函数 $K_\nu$ 的标准表示式,而式(6.7.24)则从相邻连续超几何函数之间的关系式得到,(在 Abramowitz 和 Stegun 中的式(13.4.16)和(13.4.18))。函数 $z_n$ 满足三项递推关系式(Abramowitz 和 Steagun 中方程式(13.4.15))

$$z_{n-1}(x) = b_n z_n(x) - a_{n+1} z_{n+1} \tag{6.7.25}$$

其中

$$\begin{aligned} b_n &= 2(n + x) \\ a_{n+1} &= -[(n + 1/2)^2 - \nu^2] \end{aligned} \tag{6.7.26}$$

按照得到式(5.5.18)的步骤,得到连分式 CF2

$$\frac{z_1}{z_0} = \frac{1}{b_1 +}\,\frac{a_2}{b_2 +}\cdots \tag{6.7.27}$$

由此从式(6.7.24)给出 $K_{\nu+1}/K_\nu$ 及 $K_\nu'/K_\nu$。

Temme 的正则化条件为

$$\sum_{n=0}^{\infty} C_n z_n = \left(\frac{1}{2x}\right)^{\nu+1/2} \tag{6.7.28}$$

其中

$$C_n = \frac{(-1)^n}{n!}\,\frac{\Gamma(\nu + 1/2 + n)}{\Gamma(\nu + 1/2 - n)} \tag{6.7.29}$$

注意 $C_n$ 可由下面递推式决定:

$$C_0 = 1, \quad C_{n+1} = -\frac{a_{n+1}}{n + 1} C_n \tag{6.7.30}$$

用式(6.7.28)条件求

$$S = \sum_{n=1}^{\infty} C_n \frac{z_n}{z_0} \tag{6.7.31}$$

则

$$z_0 = \left(\frac{1}{2x}\right)^{\nu+1/2} \frac{1}{1 - S} \tag{6.7.32}$$

连同式(6.7.23)得到 $K_\nu$。

Thompson 和 Barnette[4]找到一个很聪明的方法,同时求和(6.7.31)及前向求连分式 CF2值。假定连分式求值时得到

$$\frac{z_1}{z_0} = \sum_{n=0}^{\infty} \Delta h_n \tag{6.7.33}$$

其中增量 $\Delta h_n$ 可由 Steed 算法或第5.2节修正 Lentz 算法得到。$S$ 的前 $N$ 项和的逼近式为:

$$S_N = \sum_{n=1}^{N} Q_n \Delta h_n \tag{6.7.34}$$

其中

$$Q_n = \sum_{k=1}^{n} C_k q_k \tag{6.7.35}$$

$q_k$ 则由下面递推式得到

$$q_{k+1} = -(q_{k-1} - b_k q_k)/a_{k+1} \tag{6.7.36}$$

起始值 $q_0=0$,$q_1=1$,对现在情况,$S$ 收敛所需的项大约是使 CF2收敛所需项数的三倍。

$x$ 较小时,求 $K_\nu$ 和 $K_{\nu+1}$ 可用类似式(6.7.14)的级数

$$K_\nu = \sum_{k=0}^{\infty} c_k f_k \qquad K_{\nu+1} = \frac{2}{x}\sum_{k=0}^{\infty} c_k h_k \tag{6.7.37}$$

其中

$$\begin{aligned} c_k &= \frac{(x^2/4)^k}{k!} \\ h_k &= -kf_k + p_k \\ p_k &= \frac{p_{k-1}}{k-\nu} \\ q_k &= \frac{q_{k-1}}{k+\nu} \\ f_k &= \frac{kf_{k-1} + p_{k-1} + q_{k-1}}{k^2-\nu^2} \end{aligned} \tag{6.7.38}$$

递推式的初始值为

$$\begin{aligned} p_0 &= \frac{1}{2}\left(\frac{x}{2}\right)^{-\nu}\Gamma(1+\nu) \\ q_0 &= \frac{1}{2}\left(\frac{x}{2}\right)^{\nu}\Gamma(1-\nu) \\ f_0 &= \frac{\nu\pi}{\sin\nu\pi}\left[\cosh\sigma\,\Gamma_1(\nu) + \frac{\sinh\sigma}{\sigma}\ln\left(\frac{2}{x}\right)\Gamma_2(\nu)\right] \end{aligned} \tag{6.7.39}$$

$x$ 较小时的级数，CF2及正则关系式(6.7.28)都需$|\nu|\leqslant 1/2$，所以在此范围内可将 $I_\nu$ 递推到 $\nu=\mu$ 处，求得 $K_\mu$，再将 $K_\nu$ 反向递推至 $\nu$ 的初始值。

程序假定 $\nu\geqslant 0$。$\nu$ 为负数时则用反射公式

$$\begin{aligned} I_{-\nu} &= I_\nu + \frac{2}{\pi}\sin(\nu\pi)K_\nu \\ K_{-\nu} &= K_\nu \end{aligned} \tag{6.7.40}$$

注意，当 $x$ 很大时，$I_\nu \sim e^x$，$K_\nu \sim e^{-x}$，这样该函数会下溢或上溢。较好的办法是调整变量为 $e^{-x}I_\nu$ 和 $e^x k_\nu$。在式(6.7.23)略去 $e^{-x}$ 因子可保证四个量均有合适的调整范围。若使用式(6.7.37)的级数，在 $x$ 很小时要想调整四个量，则仅需对每个级数乘以 $e^x$。

```
#include <math.h>
#define EPS 1.0e-10
#define FPMIN 1.0e-30
#define MAXIT 10000
#define XMIN 2.0
#define PI 3.141592653589793

void bessik(float x, float xnu, float *ri, float *rk, float *rip, float *rkp)
```

对正数 x 和 xnu≥0，返回修正贝塞尔函数 ri=$I_\nu$，rk=$K_\nu$ 及其导数 rip=$I'_\nu$，rkp=$K'_\nu$，相对精度在 EPS 的一至两位有效数字范围之内。FPMIN 接近机器最小可表示浮点数，内部算法为双精度。只需将上面的浮点数说明改为双精度说明及将 EPS 改为$10^{-16}$，就可将整个程序改为双精度。同时需要修正函数 beschb。

```
{
    void beschb(double x, double *gam1, double *gam2, double *gampl,
        double *gammi);
    void nrerror(char error_text[]);
    int i,l,nl;
    double a,a1,b,c,d,del,del1,delh,dels,e,f,fact,fact2,ff,gam1,gam2,
        gammi,gampl,h,p,pimu,q,q1,q2,qnew,ril,ril1,rimu,rip1,ripl,
        ritemp,rk1,rkmu,rkmup,rktemp,s,sum,sum1,x2,xi,xi2,xmu,xmu2;
```

```
if (x <= 0.0 || xnu < 0.0) nrerror("bad arguments in bessik");
nl=(int)(xnu+0.5);                              nl为I的后向递推次数和K的前向递推
xmu=xnu-nl;                                         次数，xmu落在-1/2至1/2之间
xmu2=xmu*xmu;
xi=1.0/x;
xi2=2.0*xi;
h=xnu*xi;                                       由修正Lentz方法求CF1值
if (h < FPMIN) h=FPMIN;
b=xi2*xnu;
d=0.0;
c=h;
for (i=1;i<=MAXIT;i++) {
    b += xi2;
    d=1.0/(b+d);                                分母不能为零，但不必过于担心
    c=b+1.0/c;
    del=c*d;
    h=del*h;
    if (fabs(del-1.0) < EPS) break;
}
if (i > MAXIT) nrerror("x too large in bessik; try asymptotic expansion");
ril=FPMIN;                                      初始化I_ν，I'_ν，准备向后递推
ripl=h*ril;
ril1=ril;                                       存储值准备以后调整
rip1=ripl;
fact=xnu*xi;
for (l=nl;l>=1;l--) {
    ritemp=fact*ril+ripl;
    fact -= xi;
    ripl=fact*ritemp+ril;
    ril=ritemp;
}
f=ripl/ril;                                     得到未规一化的I_μ，I'_μ
if (x < XMIN) {                                 使用级数
    x2=0.5*x;
    pimu=PI*xmu;
    fact = (fabs(pimu) < EPS ? 1.0 : pimu/sin(pimu));
    d = -log(x2);
    e=xmu*d;
    fact2 = (fabs(e) < EPS ? 1.0 : sinh(e)/e);
    beschb(xmu,&gam1,&gam2,&gampl,&gammi);      求Γ1，Γ2切比雪夫数值
    ff=fact*(gam1*cosh(e)+gam2*fact2*d);        f0.
    sum=ff;
    e=exp(e);
    p=0.5*e/gampl;                              p0.
    q=0.5/(e*gammi);                            q0.
    c=1.0;
    d=x2*x2;
    sum1=p;
    for (i=1;i<=MAXIT;i++) {
        ff=(i*ff+p+q)/(i*i-xmu2);
        c *= (d/i);
        p /= (i-xmu);
        q /= (i+xmu);
        del=c*ff;
        sum += del;
        del1=c*(p-i*ff);
        sum1 += del1;
        if (fabs(del) < fabs(sum)*EPS) break;
    }
    if (i > MAXIT) nrerror("bessk series failed to converge");
    rkmu=sum;
    rk1=sum1*xi2;
} else {                                        Steed法求CF2值，因分母不能为零，
    b=2.0*(1.0+x);                                  故可行
    d=1.0/b;
    h=delh=d;
    q1=0.0;                                     (6.7.35)递推式初始化
    q2=1.0;
```

```
        a1=0.25-xmu2;
        q=c*a1;                                       (6.7.34)式第一项
        a = -a1;
        s=1.0+q*delh;
        for (i=2;i<=MAXIT;i++) {
            a -= 2*(i-1);
            c = -a*c/i;
            qnew=(q1-b*q2)/a;
            q1=q2;
            q2=qnew;
            q += c*qnew;
            b += 2.0;
            d=1.0/(b+a*d);
            delh=(b*d-1.0)*delh;
            h += delh;
            dels=q*delh;
            s += dels;
            if (fabs(dels/s) < EPS) break;
            因为 CF2 本身很快地收敛，所以只需测试和项的收敛性
        }
        if (i > MAXIT) nrerror("bessik: failure to converge in cf2");
        h=a1*h;
        rkmu=sqrt(PI/(2.0*x))*exp(-x)/s;              略去因子exp(-x)，当x<XMIN时。
        rk1=rkmu*(xmu+x+0.5-h)*xi;                       用exp(x)调整所有返回函数
    }
    rkmup=xmu*xi*rkmu-rk1;
    rimu=xi/(f*rkmu-rkmup);                           由Wronskian得到I_μ
    *ri=(rimu*ril1)/ril;                              调整初始I_ν和I'_ν
    *rip=(rimu*rip1)/ril;
    for (i=1;i<=nl;i++) {                             向前递推K_ν
        rktemp=(xmu+i)*xi2*rk1+rkmu;
        rkmu=rk1;
        rk1=rktemp;
    }
    *rk=rkmu;
    *rkp=xmu*xi*rkmu-rk1;
}
```

### 6.7.3 Airy 函数

对正数 $x$，Airy 函数定义为

$$\mathrm{Ai}(x)=\frac{1}{\pi}\sqrt{\frac{x}{3}}K_{1/3}(z) \tag{6.7.41}$$

$$\mathrm{Bi}(x)=\sqrt{\frac{x}{3}}\left[I_{1/3}(z)+I_{-1/3}(z)\right] \tag{6.7.42}$$

其中

$$z=\frac{2}{3}x^{3/2} \tag{6.7.43}$$

应用反射公式(6.7.40)，可将(6.7.42)转换成更有用的可计算形式：

$$\mathrm{Bi}(x)=\sqrt{x}\left[\frac{2}{\sqrt{3}}I_{1/3}(z)+\frac{1}{\pi}K_{1/3}(z)\right] \tag{6.7.44}$$

故 Ai、Bi 只需调用一次 **bessik** 即可求出值。

导数不能用上面表达式进行简单地微分求得，因为在 $x=0$附近可能会相消。可以使用下面等价形式：

$$\mathrm{Ai}'(x)=-\frac{x}{\pi\sqrt{3}}K_{2/3}(z)$$
$$\mathrm{Bi}'(x)=x\left[\frac{2}{\sqrt{3}}I_{2/3}(Z)+\frac{1}{\pi}K_{2/3}(z)\right] \tag{6.7.45}$$

对负变量相应公式为：

$$
\begin{aligned}
\mathrm{Ai}(-x) &= \frac{\sqrt{x}}{2}\left[J_{1/3}(z) - \frac{1}{\sqrt{3}}Y_{1/3}(z)\right] \\
\mathrm{Bi}(-x) &= -\frac{\sqrt{x}}{2}\left[\frac{1}{\sqrt{3}}J_{1/3}(z) + Y_{1/3}(z)\right] \\
\mathrm{Ai}'(-x) &= \frac{x}{2}\left[J_{2/3}(z) - \frac{1}{\sqrt{3}}Y_{2/3}(z)\right] \\
\mathrm{Bi}'(-x) &= \frac{x}{2}\left[\frac{1}{\sqrt{3}}J_{2/3}(z) - Y_{2/3}(z)\right]
\end{aligned}
\qquad (6.7.46)
$$

```
#include <math.h>
#define PI 3.1415927
#define THIRD (1.0/3.0)
#define TWOTHR (2.0*THIRD)
#define ONOVRT 0.57735027

void airy(float x, float *ai, float *bi, float *aip, float *bip)
    返回 Airy 函数 Ai(x),Bi(x)及其导数 Ai'(x),Bi'(x).
{
    void bessik(float x, float xnu, float *ri, float *rk, float *rip,
        float *rkp);
    void bessjy(float x, float xnu, float *rj, float *ry, float *rjp,
        float *ryp);
    float absx,ri,rip,rj,rjp,rk,rkp,rootx,ry,ryp,z;

    absx=fabs(x);
    rootx=sqrt(absx);
    z=TWOTHR*absx*rootx;
    if (x > 0.0) {
        bessik(z,THIRD,&ri,&rk,&rip,&rkp);
        *ai=rootx*ONOVRT*rk/PI;
        *bi=rootx*(rk/PI+2.0*ONOVRT*ri);
        bessik(z,TWOTHR,&ri,&rk,&rip,&rkp);
        *aip = -x*ONOVRT*rk/PI;
        *bip=x*(rk/PI+2.0*ONOVRT*ri);
    } else if (x < 0.0) {
        bessjy(z,THIRD,&rj,&ry,&rjp,&ryp);
        *ai=0.5*rootx*(rj-ONOVRT*ry);
        *bi = -0.5*rootx*(ry+ONOVRT*rj);
        bessjy(z,TWOTHR,&rj,&ry,&rjp,&ryp);
        *aip=0.5*absx*(ONOVRT*ry+rj);
        *bip=0.5*absx*(ONOVRT*rj-ry);
    } else {                    Case x = 0.
        *ai=0.35502805;
        *bi=(*ai)/ONOVRT;
        *aip = -0.25881940;
        *bip = -(*aip)/ONOVRT;
    }
}
```

## 6.7.4 球面贝塞尔函数

对整数 $n$,球面贝塞尔函数定义为

$$
j_n(x) = \sqrt{\frac{\pi}{2x}}J_{n+(1/2)}(x)
$$

$$y_n(x)=\sqrt{\frac{\pi}{2x}}Y_{n+(1/2)}(x) \tag{6.7.47}$$

只需调用一次 **bessjy** 即可求其值，导数也可由式(6.7.47)的微分求出。

注意，(6.7.3)式的连分式 CF2当 $\nu=1/2$ 时仅存在第一项。这样对球面函数，按程序 **bessjy** 可找到很简便的算法。常常只需对 $j_n$ 递推到 $n=0$，由 CF2的第一项确定 $p$ 和 $q$，然后对 $y_n$ 向前递推。在 $x=0$处，不需另外特殊的级数。程序 **bessjy** 已经完全有效，没有必要劳神费力地为球面贝塞尔函数再去设计一个独立的程序。

```
#include <math.h>
#define RTPIO2 1.2533141

void sphbes(int n, float x, float *sj, float *sy, float *sjp, float *syp)
    对整数n返回球面贝塞尔函数 j_n(x),y_n(x)及其导数 j_n'(x),y_n'(x)。
{
    void bessjy(float x, float xnu, float *rj, float *ry, float *rjp, float *ryp);
    void nrerror(char eror_text[]);
    float factor,order,rj,rjp,ry,ryp;

    if(n<0 || x<=0.0) nrerror("bad arguments in sphbes");
    order=n+0.5;
    bessjy(x,order,&rj,&ry,&rjp,&ryp);
    factor=RTPIO2/sqrt(x);
    *sj=factor*rj;
    *sy=factor*ry;
    *sjp=factor*rjp-(*sj)/(2.0*x);
    *syp=factor*ryp-(*sy)/(2.0*x);
}
```

参考文献和进一步读物：


Barnett, A.R., Feng, D.H., Steed, J.W., and Goldfarb, L.J.B. 1974, *Computer Physics Communications*, vol. 8, pp. 377～395. [1]

Temme, N.M. 1976, *Journal of Computational Physics*, vol. 21, pp. 343～350. [2]; 1975, *op. cit.*, vol. 19, pp. 324～337. [3]

Thompson, I.J., and Barnett, A.R. 1987, *Computer Physics, communications*, vol. 47, pp. 245～257. [4]


## 6.8 球面调和函数

球面调和函数出现在大量物理问题中，例如，波动方程或拉普拉斯(Laplace)方程，就是通过在球面坐标下用变量分离法解决的。球面调和函数 $Y_{lm}(\theta,\varphi)$，$-l\leqslant m\leqslant l$，是球面上两个坐标 $\theta,\varphi$ 的函数。

对于不同 $l,m$ 球面调和函数是正交的，且经过规一化，使之在球面上的平方积分为单位1：

$$\int_0^{2\pi}d\varphi\int_{-1}^{1}d(\cos\theta)Y_{l'm'}{}^*(\theta,\varphi)Y_{lm}(\theta,\varphi)=\delta_{l'l}\delta_{m'm} \tag{6.8.1}$$

其中星号(*)表示复共轭。

数学上，联系球面调和函数与**连带的勒让德**(Legendre)**多项式**的关系式为：

$$Y_{lm}(\theta,\varphi)=\sqrt{\frac{2l+1}{4\pi}\frac{(l-m)!}{(l+m)!}}P_l^m(\cos\theta)e^{im\varphi} \tag{6.8.2}$$

应用关系式

$$Y_{l,-m}(\theta,\varphi)=(-1)^m Y_{lm}^*(\theta,\varphi) \tag{6.8.3}$$

我们总能将一个球面调用和函数，与一个 $m\geq 0$的连带的勒让德多项式联系起来。取 $x=\cos\theta$，这种多项式由普通勒让德多项式定义成(参见第4.5节及第5.4节)

$$P_l^m(x)=(-1)^m(1-x^2)^{m/2}\frac{d^m}{dx^m}P_l(x) \tag{6.8.4}$$

前面几项连带的勒让德多项式及与其对应的规一化后的球面调和函数，如表6.8.1所示。

**表6.8.1 连带勒让德多项式和球面调和函数的对应关系**

| | |
|---|---|
| $P_0^0(x)=1$ | $Y_{00}=\sqrt{\frac{1}{4\pi}}$ |
| $P_1^1(x)=-(1-x^2)^{1/2}$ | $Y_{11}=-\sqrt{\frac{3}{8\pi}}\sin\theta e^{i\varphi}$ |
| $P_1^0(x)=x$ | $Y_{10}=\sqrt{\frac{3}{4\pi}}\cos\theta$ |
| $P_2^2(x)=3(1-x^2)$ | $Y_{22}=\frac{1}{4}\sqrt{\frac{15}{2\pi}}\sin^2\theta e^{2i\varphi}$ |
| $P_2^1(x)=-3(1-x^2)^{1/2}x$ | $Y_{21}=-\sqrt{\frac{15}{8\pi}}\sin\theta\cos\theta e^{i\varphi}$ |
| $P_2^0(x)=\frac{1}{2}(3x^2-1)$ | $Y_{20}=\sqrt{\frac{5}{4\pi}}(\frac{3}{2}\cos^2\theta-\frac{1}{2})$ |

(6.8.5)

数值计算连带勒让德多项方法很多，但许多方法都不很好。例如，有显式表达式如

$$P_l^m(x)=\frac{(-1)^m(l+m)!}{2^m m!(l-m)!}(1-x^2)^{m/2}\left[1-\frac{(l-m)(m+l+1)}{1!(m+1)}\left(\frac{1-x}{2}\right)\right.$$
$$\left.+\frac{(l-m)(l-m-1)(m+l+1)(m+l+2)}{2!(m+1)(m+2)}\left(\frac{1-x}{2}\right)^2-\cdots\right] \tag{6.8.6}$$

其中多项式只到含有$(1-x)^{l-m}$的项。(关于此式及相关公式参见[1])。这种方法不能叫人满意，因为逐项符号交替，因此求多项式值时，符号相反的连续项可巧妙地消去。$l$ 很大时，多项式中单独项却比和式大得多，而且精度很差。

实用中，依精度要求，$l$ 达到6或8时式(6.8.6)按单精度(32位)使用；$l$ 达到15或18时，按双精度(64位)使用。因而希望能找到一种更有力的计算方法，如下述：

连带勒让德多项式满足很多递推关系[1-2]。有仅对l递推的，有仅对 $m$ 递推的，也有同时对l和m递推的。大多数对 $m$ 的递推不稳定，不适于数值计算。下面这个递推式对 l 的递推却是稳定的(试与式(5.5.1)比较)：

$$(l-m)P_l^m=x(2l-1)P_{l-1}^m-(l+m-1)P_{l-2}^m \tag{6.8.7}$$

这个式子很有用，因为对初始值存在闭形表达式：

$$P_m^m=(-1)^m(2m-1)!!(1-x^2)^{m/2} \tag{6.8.8}$$

(记号 $n!!$ 表示小于等于 $n$ 的所有奇整数的乘积。)对式(6.8.7),令 $l=m+1$ 及 $P_{m-1}^{m}=0$,得到

$$P_{m+1}^{m}=x(2m+1)P_{m}^{m} \tag{6.8.9}$$

式(6.8.7)中对一般 $l$ 所要求的两个初始值可由式(6.8.8)和(6.8.9)给出。实现该算法的程序函数为

```
#include <math.h>

float plgndr(int l, int m, float x)
    计算连带勒让德多项式 P_l^m(x),其中 m 和 l 为整数,满足0≤m≤l,x 满足-1≤x≤1。
{
    void nrerror(char error_text[]);
    float fact,pll,pmm,pmmp1,somx2;
    int i,ll;

    if (m < 0 || m > l || fabs(x) > 1.0)
        nrerror("Bad arguments in routine plgndr");
    pmm=1.0;                        计算 P_m^m
    if (m > 0) {
        somx2=sqrt((1.0-x)*(1.0+x));
        fact=1.0;
        for (i=1;i<=m;i++) {
            pmm *= -fact*somx2;
            fact += 2.0;
        }
    }
    if (l == m)
        return pmm;
    else {                          计算 P_{m+1}^m
        pmmp1=x*(2*m+1)*pmm;
        if (l == (m+1))
            return pmmp1;
        else {                      计算 P_l^m, l > m + 1
            for (ll=m+2;ll<=l;ll++) {
                pll=(x*(2*ll-1)*pmmp1-(ll+m-1)*pmm)/(ll-m);
                pmm=pmmp1;
                pmmp1=pll;
            }
            return pll;
        }
    }
}
```

参考文献和进一步读物:


Magnus, W., and Oberhettinger, F. 1949, *Formulas and Theorems for the Functions of Mathematical Physics* (New York: Chelsea), pp. 54ff. [1]

Abramowitz, M., and Stegun, I. A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, vol. 55 Chapter 8. [2]


## 6.9 菲涅耳积分、余弦和正弦积分

### 6.9.1 菲涅耳(Fresnel)积分

定义两个菲涅耳积分为

$$C(x)=\int_0^x \cos\left(\frac{\pi}{2}t^2\right)dt,\qquad S(x)=\int_0^x \sin\left(\frac{\pi}{2}t^2\right)dt \tag{6.9.1}$$

$x$ 较小时，对任意精度要求，计算函数值最方便的方法是幂级函数；$x$ 较大时，则用连分式。级数为

$$\begin{aligned} C(x) &= x-\left(\frac{\pi}{2}\right)^2\frac{x^5}{5\cdot 2!}+\left(\frac{\pi}{2}\right)^4\frac{x^9}{9\cdot 4!}-\cdots \\ S(x) &= \left(\frac{\pi}{2}\right)\frac{x^3}{3\cdot 1!}-\left(\frac{\pi}{2}\right)^3\frac{x^7}{7\cdot 3!}+\left(\frac{\pi}{2}\right)^5\frac{x^{11}}{11\cdot 5!}-\cdots \end{aligned} \tag{6.9.2}$$

存在一种复连分式，可同时求出 $C(x)$ 和 $S(x)$。

$$C(x)+iS(x)=\frac{1-i}{2}\operatorname{erf} z,\quad z=\frac{\sqrt{\pi}}{2}(1-i)x \tag{6.9.3}$$

其中

$$\begin{aligned} e^{z^2}\operatorname{erfc} z &= \frac{1}{\sqrt{\pi}}\left(\frac{1}{z+}\,\frac{1/2}{z+}\,\frac{1}{z+}\,\frac{3/2}{z+}\,\frac{2}{z+}\cdots\right) \\ &= \frac{2z}{\sqrt{\pi}}\left(\frac{1}{2z^2+1-}\,\frac{1\cdot 2}{2z^2+5-}\,\frac{3\cdot 4}{2z^2+9-}\cdots\right) \end{aligned} \tag{6.9.4}$$

上式第二行中将连分式由通常形式转换为偶次形式（见第5.2节），这样收敛快两倍，必须小心，不要在 $x$ 太大时，求交错级数式(6.9.2)的值。从级数项中可以看出，$x=1.5$ 是切换到使用连分式比较合适的点。

注意，当 $x$ 很大时

$$C(x)\sim\frac{1}{2}+\frac{1}{\pi x}\sin\left(\frac{\pi}{2}x^2\right),\quad S(x)\sim\frac{1}{2}-\frac{1}{\pi x}\cos\left(\frac{\pi}{2}x^2\right) \tag{6.9.5}$$

这样，程序 **frenel** 的精度可能会受到 $x$ 较大时库函数中正弦和余弦精度的影响。

```
#include <math.h>
#include "complex.h"
#define EPS 6.0e-8                 相对误差
#define MAXIT 100                  允许最大迭代次数
#define FPMIN 1.0e-30              最小可表示的浮点数的邻近数
#define XMIN 1.5                   使用级数和连分式之间的区分线
#define PI 3.1415927
#define PIBY2 (PI/2.0)
#define TRUE 1
#define ONE Complex(1.0,0.0)

void frenel(float x, float *s, float *c)     对所有实数 x 计算菲涅耳积分 S(x)和 C(x)
void nrerror(char error_text[]);
int k,n,odd;
float a,ax,fact,pix2,sign,sum,sumc,sums,term,test;
fcomplex b,cc,d,h,del,cs;

ax=fabs(x);
if (ax < sqrt(FPMIN)) {                      特殊情况：避免因为下溢而使收敛测试失败
    *s=0.0;
    *c=ax;
} else if (ax <= XMIN) {                     同时求两级数值
    sum=sums=0.0;
    sumc=ax;
    sign=1.0;
    fact=PIBY2*ax*ax;
    odd=TRUE;
```

```
        term=ax;
        n=3;
        for (k=1;k<=MAXIT;k++) {
            term *= fact/k;
            sum += sign*term/n;
            test=fabs(sum)*EPS;
            if (odd) {
                sign = -sign;
                sums=sum;
                sum=sumc;
            } else {
                sumc=sum;
                sum=sums;
            }
            if (term < test) break;
            odd=!odd;
            n += 2;
        }
        if (k > MAXIT) nrerror("series failed in frenel");
        *s=sums;
        *c=sumc;
    } else {                                由修正Lentz方法求连分式值
        pix2=PI*ax*ax;
        b=Complex(1.0,-pix2);
        cc=Complex(1.0/FPMIN,0.0);
        d=h=Cdiv(ONE,b);
        n = -1;
        for (k=2;k<=MAXIT;k++) {
            n += 2;
            a = -n*(n+1);
            b=Cadd(b,Complex(4.0,0.0));
            d=Cdiv(ONE,Cadd(RCmul(a,d),b));    分母不能为零
            cc=Cadd(b,Cdiv(Complex(a,0.0),cc));
            del=Cmul(cc,d);
            h=Cmul(h,del);
            if (fabs(del.r-1.0)+fabs(del.i) < EPS) break;
        }
        if (k > MAXIT) nrerror("cf failed in frenel");
        h=Cmul(Complex(ax,-ax),h);
        cs=Cmul(Complex(0.5,0.5),
            Csub(ONE,Cmul(Complex(cos(0.5*pix2),sin(0.5*pix2)),h)));
        *c=cs.r;
        *s=cs.i;
    }
    if (x < 0.0) {                          应用反对称性
        *c = -(*c);
        *s = -(*s);
    }
}
```

## 6.9.2 余弦和正弦积分

余弦和正弦积分定义为

$$
\begin{aligned}
\mathrm{Ci}(x) &= \gamma + \ln x - \int_0^x \frac{\cos t - 1}{t} dt \\
\mathrm{Si}(x) &= \int_0^x \frac{\sin t}{t} dt
\end{aligned}
\tag{6.9.6}
$$

其中 $\gamma \approx 0.5772\cdots$ 为欧拉常数，因

$$
\mathrm{Si}(-x) = -\mathrm{Si}(x), \qquad \mathrm{Ci}(-x) = \mathrm{Ci}(x) - i\pi \tag{6.9.7}
$$

故仅需对 $x>0$找出求函数的方法。

我们还是可以用,幂级数和复连分式相结合的方法求此函数值。级数为

$$\begin{aligned} \mathrm{Si}(x) &= x - \frac{x^3}{3\cdot 3!} + \frac{x^5}{5\cdot 5!} - \cdots \\ \mathrm{Ci}(x) &= \gamma + \ln x + \left(-\frac{x^2}{2\cdot 2!} + \frac{x^4}{4\cdot 4!} - \cdots\right) \end{aligned} \tag{6.9.8}$$

指数积分 $E_1(ix)$的连分式为

$$\begin{aligned} E_1(ix) &= -\mathrm{Ci}(x) + i[\mathrm{Si}(x) - \pi/2] \\ &= e^{-ix}\left(\frac{1}{ix+}\frac{1}{1+}\frac{1}{ix+}\frac{2}{1+}\frac{2}{ix+}\cdots\right) \\ &= e^{-ix}\left(\frac{1}{1+ix-}\frac{1^2}{3+ix-}\frac{2^2}{5+ix-}\cdots\right) \end{aligned} \tag{6.9.9}$$

上式中最后一行是连分式的偶形式,对同样计算量它收敛可快两倍。现在从交错级数到连分式的转换点为 $x=2$ 。当 $x$ 很大时,菲涅耳积分的精度会受到 sine 和 cosine 的程序精度的限制。

```
#include <math.h>
#include "complex.h"
#define EPS 6.0e-8              相对误差或ci(x)接近零时的绝对误差
#define EULER 0.57721566        欧拉常数γ
#define MAXIT 100               允许最大迭代数
#define PIBY2 1.5707963         π/2
#define FPMIN 1.0e-30           最小可表示的浮点数的邻近数
#define TMIN 2.0                采用级数和连分式之间的划分线
#define TRUE 1
#define ONE Complex(1.0,0.0)

void cisi(float x, float *ci, float *si)
    计算余弦和正弦积分 Ci(x)和 Si(x)。Ci(0)返回一个大负数,不给出错误信息。x<0时程序返回 Ci(-x),必须用户
    自己另加(-iπ)因子。
{
    void nrerror(char error_text[]);
    int i,k,odd;
    float a,err,fact,sign,sum,sumc,sums,t,term;
    fcomplex h,b,c,d,del;

    t=fabs(x);
    if (t == 0.0) {                         特殊情况
        *si=0.0;
        *ci = -1.0/FPMIN;
        return;
    }
    if (t > TMIN) {                         由修正Lentz方法求连分式
        b=Complex(1.0,t);
        c=Complex(1.0/FPMIN,0.0);
        d=h=Cdiv(ONE,b);
        for (i=2;i<=MAXIT;i++) {
            a = -(i-1)*(i-1);
            b=Cadd(b,Complex(2.0,0.0));
            d=Cdiv(ONE,Cadd(RCmul(a,d),b));   分母不能为零
            c=Cadd(b,Cdiv(Complex(a,0.0),c));
            del=Cmul(c,d);
            h=Cmul(h,del);
            if (fabs(del.r-1.0)+fabs(del.i) < EPS) break;
        }
```

```
        if (i > MAXIT) nrerror("cf failed in cisi");
        h=Cmul(Complex(cos(t),-sin(t)),h);
        *ci = -h.r;
        *si=PIBY2+h.i;
    } else {                                    同时求两级数值
        if (t < sqrt(FPMIN)) {                  特殊情况：避免因为下溢出现而使收敛
            sumc=0.0;                               检测失败
            sums=t;
        } else {
            sum=sums=sumc=0.0;
            sign=fact=1.0;
            odd=TRUE;
            for (k=1;k<=MAXIT;k++) {
                fact *= t/k;
                term=fact/k;
                sum += sign*term;
                err=term/fabs(sum);
                if (odd) {
                    sign = -sign;
                    sums=sum;
                    sum=sumc;
                } else {
                    sumc=sum;
                    sum=sums;
                }
                if (err < EPS) break;
                odd=!odd;
            }
            if (k > MAXIT) nrerror("maxits exceeded in cisi");
        }
        *si=sums;
        *ci=sumc+log(t)+EULER;
    }
    if (x < 0.0) *si = -(*si);
}
```

## 6.10 道生积分

道生(Dawson)积分定义为

$$F(x) = e^{-x^2}\int_0^x e^{t^2}dt \tag{6.10.1}$$

此函数与复误差函数关系为

$$F(z) = \frac{i\sqrt{\pi}}{2}e^{-z^2}[1 - \mathrm{erfc}(-iz)] \tag{6.10.2}$$

Rybicki[1]找到一个极其接近的表达

$$F(z) = \lim_{h\to 0}\frac{1}{\sqrt{\pi}}\sum_{n\ \mathrm{odd}}\frac{e^{-(z-nh)^2}}{n} \tag{6.10.3}$$

式(6.10.3)惊人之处在于，其精度在 $h$ 较小时呈指数增长，所以一般的 $h$ 值(级数有相当快的收敛速度)就能得到相当精确的逼近结果。

稍后在第13.11节中我们将讨论导出式(6.10.3)的理论，它作为傅里叶方法一个有趣的应用。这里仅给出基于公式的实现程序。

将和式的下标移至指数项的最大项附近，是最为方便。所以定义 $n_0$ 为最接近 $x/h$ 的偶整数，$x_0 \equiv n_0 h$，$x' \equiv x - x_0$，$n' \equiv n - n_0$，则有

$$F(x) \approx \frac{1}{\sqrt{\pi}}\sum_{\substack{n'=-N \\ n'\ \mathrm{odd}}}^{N}\frac{e^{-(x'-n'h)^2}}{n'+n_0} \tag{6.10.4}$$

$h$ 足够小并 $N$ 充分大时，这个近似等号是足够精确的。若注意到下式

$$e^{-(x'-n'h)^2} = e^{-x'^2} e^{-(n'h)^2} (e^{2x'h})^{n'} \tag{6.10.5}$$

则式(6.10.4)的计算速度还可以大大加快。第一个因子只需计算一次，第二项则为可存储的常数数组，第三项可递推地计算，因此只需计算两个指数项。另外还有一个优点就是，将求和号按 $n'$ 分成正值和负值两部分时，系数 $e^{-(n'h)^2}$ 具有对称性。

下面程序中选定 $h=0.4$，$N=11$。因为求和的对称性及 $n$ 限定为奇数，所以 do 循环限定为1～6。在浮点型版本中，此程序计算结果的精度大约为$2\times10^{-7}$。$x=0$ 附近，$F(x)$不存在。为保证精度，程序中在$|x|<0.2$ 时，用 $F(x)$的幂级数[2]求值。

```
#include <math.h>
#include "nrutil.h"
#define NMAX 6
#define H 0.4
#define A1  (2.0/3.0)
#define A2  0.4
#define A3  (2.0/7.0)

float dawson(float x)      对任何实数 x 返回道生积分 F(x) = exp(-x²)∫₀ˣ exp(t²)dt
{
    int i,n0;
    float d1,d2,e1,e2,sum,x2,xp,xx,ans;
    static float c[NMAX+1];
    static int init = 0;           若需初始化，标志为0，否则为 1

    if (init == 0) {
        init=1;
        for (i=1;i<=NMAX;i++) c[i]=exp(-SQR((2.0*i-1.0)*H));
    }
    if (fabs(x) < 0.2) {           利用级数扩展
        x2=x*x;
        ans=x*(1.0-A1*x2*(1.0-A2*x2*(1.0-A3*x2)));
    } else {                       利用取样定理表达式
        xx=fabs(x);
        n0=2*(int)(0.5*xx/H+0.5);
        xp=xx-n0*H;
        e1=exp(2.0*xp*H);
        e2=e1*e1;
        d1=n0+1;
        d2=d1-2.0;
        sum=0.0;
        for (i=1;i<=NMAX;i++,d1+=2.0,d2-=2.0,e1*=e2)
            sum += c[i]*(e1/d1+1.0/(d2*e1));
        ans=0.5641895835*SIGN(exp(-xp*xp),x)*sum;      常数为1/√π
    }
    return ans;
}
```

还有其它一些计算道生积分的方法，见[2,3]。

参考文献和进一步读物：


Rybicki, G.B. 1989, *Computers in Physics*, vol. 3, no. 2, pp. 85～87. [1]

Cody, W.J., Pociorek, K.A., and Thatcher, H.C. 1970, *Mathematics of Computation*, vol. 24, pp. 171～178. [2]

McCabe, J.H. 1974, *Mathematics of Computation*, vol. 28, pp. 811~816. [3]

## 6.11 椭圆积分和雅可比椭圆函数

在很多应用中要用到椭圆积分，因为任何形式如(6.11.1)式的积分都可由椭圆积分求得：

$$\int R(t,s)dt \tag{6.11.1}$$

其中 $R$ 为 $t,s$ 的有理函数，$s$ 为 $t$ 的三次或四次多项式的平方根。一般文献[1]中描述了如何进行简化，这个最初工作是勒让德完成的。勒让德证明了只需要三个基本椭圆函数。其中最简单的一种是

$$I_1 = \int_y^x \frac{dt}{\sqrt{(a_1 + b_1 t)(a_2 - b_2 t)(a_3 - b_3 t)(a_4 + b_4 t)}} \tag{6.11.2}$$

上式中，将 $s^2$的四次方写成因子相乘形式。标准积分表中，积分限总有一个是四次方的零点，另外一个积分限则是靠近下一个零点，这样在积分区间内没有奇异点。求积分 $I_1$，可将积分区间$[y,x]$分成子区间，每个区间的起点和终点都是奇异点。这样，只需区分将四个零点(按照阶的大小)分别作为积分上下限的八种情况。此外，当式(6.11.2)中有一个 $b$ 为零时，四次方简化变为三次方，最大或最小奇异点移至$\pm\infty$处，这样会产生多于八种情况(实际上，有些为前八种情况的特例。)根据第一类勒让德标准椭圆积分(下面将要定义)，一般将16种情况制成表。积分变量 $t$ 改变时，四次方的零点映射到实轴上的标准位置。这样只需两个无量纲的参量就可为勒让德积分制表。然而勒让德的表示形式掩盖了在根置换下原积分式(6.11.2)的对称性。以后我们还会回过头来讨论勒让德表示。首先，还是先做下面事：

卡尔森(Carlson)[3]给出一种新的第一类椭圆标准积分的定义：

$$R_F(x,y,z) = \frac{1}{2}\int_0^{\infty} \frac{dt}{\sqrt{(t+x)(t+y)(t+z)}} \tag{6.11.3}$$

其中 $x,y$ 非负且最多有一个零点。通过将积分区间标准化，可保留零点的置换对称性。(Weirstrass 的正则形式也具有此性质。)卡尔森首先证明了，若 $x$ 或 $y$ 是式(6.11.2)四次多项式的一个零点，则积分 $I_1$可根据 $R_F$ 写成在剩余三个零点置换下仍对称的形式。一般情况下若 $x$ 或 $y$ 都不为零点，则两个这样的 $R_F$ 函数可由加法定理合成一个，导出下面的基本公式：

$$I_1 = 2R_F(U_{12}^2, U_{13}^2, U_{14}^2) \tag{6.11.4}$$

其中

$$U_{ij} = (X_i X_j X_k Y_m + Y_i Y_j X_k X_m)/(x - y) \tag{6.11.5}$$

$$X_i = (a_i + b_i x)^{1/2}, \quad Y_i = (a_i + b_i y)^{1/2} \tag{6.11.6}$$

$i,j,k,m$ 可由1,2,3,4置换。一个简便些的计算方法是：

$$\begin{aligned} U_{13}^2 &= U_{12}^2 - (a_1 b_4 - a_4 b_1)(a_2 b_3 - a_3 b_2) \\ U_{14}^2 &= U_{12}^2 - (a_1 b_3 - a_3 b_1)(a_2 b_4 - a_4 b_2) \end{aligned} \tag{6.11.7}$$

其中各个 $U$ 分别对应于，将四个零点分成一对一对的三种方式，所以 $I_1$在零点置换下有明显的对称性。因此，当有一个积分限为零时，式(6.11.4)重新产生所有16种情况，也包括了上下根都不为零的情况。

这样卡尔森函数允许有任何积分区间，并且对于积分区间中，被积函数的分支点的位置也是任意的。为了处理第二和第三类椭圆积分，卡尔森定义第三类标准积分为

$$R_J(x,y,z,p) = \frac{3}{2}\int_0^\infty \frac{dt}{(t+p)\sqrt{(t+x)(t+y)(t+z)}} \tag{6.11.8}$$

它对于 $x,y,z$ 对称。当两个参量相等时的退化形式可表示为

$$R_D(x,y,z) = R_J(x,y,z,z) \tag{6.11.9}$$

它对 $x,y,z$ 对称。可用函数 $R_D$ 取代第二类勒让德积分。$R_F$ 的退化形式表示为：

$$R_C(x,y) = R_F(x,y,y) \tag{6.11.10}$$

其中包含对数，反三角和反双曲线函数。

卡尔森[4-7]根据式(6.11.1)中，四次项的线性因子指数给出了积分表。例如，指数为 $(\frac{1}{2},\frac{1}{2},\frac{1}{2},-\frac{3}{2})$ 的积分，可表示为 $R_D$ 的一个单积分，它说明了 Gradshteyn 和 Ryzhik[2]列举的144种情况。

将椭圆积分降为标准型时的一些实用性细节，如处理复共轭零点，可参见卡尔森的论文[4-7]。

现在，讨论椭圆积分的数值算法。通常使用高斯或兰登(Landen)变换。降阶变换可将勒让德积分的模数 $k$ 减少而趋于零，而增阶变换则可增大而趋于1。在这些极限位置，函数可解析地表达。这些方法以四次方收敛，对第一类和第二类求积分很有效，而对第三类积分的某些情况可能不太精确。卡尔森算法[8,10]则对三类积分提供一种统一的方法，而没有明显的消去律。

算法的关键是加倍定理：

$$\begin{aligned} R_F(x,y,z) &= 2R_F(x+\lambda, y+\lambda, z+\lambda) \\ &= R_F\left(\frac{x+\lambda}{4},\frac{y+\lambda}{4},\frac{z+\lambda}{4}\right) \end{aligned} \tag{6.11.11}$$

其中

$$\lambda = (xy)^{1/2} + (xz)^{1/2} + (yz)^{1/2} \tag{6.11.12}$$

仅须变换积分变量即可证明此定理[11]。式(6.11.11)可反复迭代，直到 $R_F$ 中的变量几乎相等为止。变量相等时有：

$$R_F(x,x,x) = x^{-1/2} \tag{6.11.13}$$

若变量足够接近，则可由式(6.11.13)固定的泰勒(Taylor)展开式中，第五次项来求函数值。算法中迭代部分只是线性地收敛，误差在每次迭代中最终按因子 $4^6=4096$ 减少。一般只需迭代两到三次，若变量初始值相差很大，则可能需要迭代 6～7 次。下面列出 $R_F$ 的算法，其它情况可参阅卡尔森的文章[9]。

第一步：对 $n=0,1,2\cdots$ 计算

$$\mu_n = (x_n + y_n + z_n)/3$$

$$X_n = 1-(x_n/\mu_n),\quad Y_n = 1-(y_n/\mu_n),\quad Z_n = 1-(z_n/\mu_n)$$

$$\varepsilon_n = max(|X_n|, |Y_n|, |Z_n|)$$

若 $\varepsilon_n <$ tol 到第二步，否则计算

$$\lambda_n = (x_n y_n)^{1/2} + (x_n z_n)^{1/2} + (y_n z_n)^{1/2}$$

$$x_{n+1} = (x_n + \lambda_n)/4,\quad y_{n+1} = (y_n + \lambda_n)/4,\quad z_{n+1} = (z_n + \lambda_n)/4$$

并在此步内循环。

第二步：计算

$$E_2 = X_nY_n - Z_n^2,\qquad E_3 = X_nY_nZ_n$$

$$R_F = (1 - \frac{1}{10}E_2 + \frac{1}{14}E_3 + \frac{1}{24}E_2^2 - \frac{3}{44}E_2E_3)/(\mu_n)^{1/2}$$

有时候 $R_J$ 中的变量 $p$ 或 $R_C$ 中的变量 $y$ 为负数，而且需要积分的柯西主值。这时可由下面公式很容易求得：

$$\begin{aligned} &R_J(x,y,z,p) = \\ &\quad [(\gamma - y)R_J(x,y,z,\gamma) - 3R_F(x,y,z) + 3R_C(xz/y, p\gamma/y)]/(y - p) \end{aligned} \tag{6.11.14}$$

其中

$$\gamma = y + \frac{(x-y)(y-z)}{y-p} \tag{6.11.15}$$

$p$ 为负数时上式为正数，且

$$R_C(x,y) = \left(\frac{x}{x-y}\right)^{1/2} R_C(x-y, -y) \tag{6.11.16}$$

$p<0$ 的某些地方，$R_J$ 的柯西主值可能为零，因此式(6.11.14)在零附近可能不精确。

```
#include <math.h>
#include "nrutil.h"
#define ERRTOL 0.08
#define TINY 1.5e-38
#define BIG 3.0e37
#define THIRD (1.0/3.0)
#define C1   (1.0/24.0)
#define C2   0.1
#define C3   (3.0/44.0)
#define C4   (1.0/14.0)

float rf(float x, float y, float z)
```

计算第一类 Carlson 椭圆积分 $R_F(x,y,z)$。$x,y,z$ 必须为非负，且至多只能有一个为零，TINY 必须至少为机器下溢限的五倍，BIG 则必须最多为机器上溢极限的五分之一。

```
{
    float alamb,ave,delx,dely,delz,e2,e3,sqrtx,sqrty,sqrtz,xt,yt,zt;

    if (FMIN(FMIN(x,y),z) < 0.0 || FMIN(FMIN(x+y,x+z),y+z) < TINY ||
        FMAX(FMAX(x,y),z) > BIG)
            nrerror("invalid arguments in rf");
    xt=x;
    yt=y;
    zt=z;
    do {
        sqrtx=sqrt(xt);
        sqrty=sqrt(yt);
        sqrtz=sqrt(zt);
        alamb=sqrtx*(sqrty+sqrtz)+sqrty*sqrtz;
        xt=0.25*(xt+alamb);
        yt=0.25*(yt+alamb);
        zt=0.25*(zt+alamb);
        ave=THIRD*(xt+yt+zt);
        delx=(ave-xt)/ave;
        dely=(ave-yt)/ave;
        delz=(ave-zt)/ave;
    } while (FMAX(FMAX(fabs(delx),fabs(dely)),fabs(delz)) > ERRTOL);
    e2=delx*dely-delz*delz;
    e3=delx*dely*delz;
    return (1.0+(C1*e2-C2-C3*e3)*e2+C4*e3)/sqrt(ave);
}
```

对单精度要求(7 位有效数字)的情况，允许的误差精度参数取为 0.08 已经足够了。因为误差尺度为 $\epsilon_{\text{许}}^{6}$，可以看出 0.0025 将产生双精度(16 位有效数字)，并且最多需要两到三次迭代。因为对于其它椭圆函数，第六阶截尾误差的系数是不同的，所以 $R_C$ 算法中允许的误差值改为 0.04 和 0.0012，$R_J$，$R_D$ 中分别改为 0.05 和 0.0015。$R_C$ 的算法除了用于结合某些基本函数外，也反复用于计算 $R_J$ 中。

在 C 语言的实现程序中，根据两个依赖于机器的常数(TINY 和 BIG)来检测输入变量，以防止计算中出现上溢和下溢。对应于机器的最小值为 $3\times10^{-39}$ 和最大值为 $1.7\times10^{38}$，我们较保守地选取 TINY 和 BIG 值。应用下面给出的齐次性关系式(6.11.22)可以扩展允许的变量值的范围。

```c
#include <math.h>
#include "nrutil.h"
#define ERRTOL 0.05
#define TINY 1.0e-25
#define BIG 4.5e21
#define C1 (3.0/14.0)
#define C2 (1.0/6.0)
#define C3 (9.0/22.0)
#define C4 (3.0/26.0)
#define C5 (0.25*C3)
#define C6 (1.5*C4)

float rd(float x, float y, float z)
    计算第二类卡尔森椭圆积分 R_D(x,y,z)。x,y 必须非负，且最多一个为零，z 必须为正值，TINY 必须至少是机器上
    溢极限的负2/3次幂的2倍，BIG 必须最多是0.1×ERROL 再乘以机器下溢极限负2/3次幂。
{
    float alamb,ave,delx,dely,delz,ea,eb,ec,ed,ee,fac,sqrtx,sqrty,
        sqrtz,sum,xt,yt,zt;

    if (FMIN(x,y) < 0.0 || FMIN(x+y,z) < TINY || FMAX(FMAX(x,y),z) > BIG)
        nrerror("invalid arguments in rd");
    xt=x;
    yt=y;
    zt=z;
    sum=0.0;
    fac=1.0;
    do {
        sqrtx=sqrt(xt);
        sqrty=sqrt(yt);
        sqrtz=sqrt(zt);
        alamb=sqrtx*(sqrty+sqrtz)+sqrty*sqrtz;
        sum += fac/(sqrtz*(zt+alamb));
        fac=0.25*fac;
        xt=0.25*(xt+alamb);
        yt=0.25*(yt+alamb);
        zt=0.25*(zt+alamb);
        ave=0.2*(xt+yt+3.0*zt);
        delx=(ave-xt)/ave;
        dely=(ave-yt)/ave;
        delz=(ave-zt)/ave;
    } while (FMAX(FMAX(fabs(delx),fabs(dely)),fabs(delz)) > ERRTOL);
    ea=delx*dely;
    eb=delz*delz;
    ec=ea-eb;
    ed=ea-6.0*eb;
    ee=ed+ec+ec;
    return 3.0*sum+fac*(1.0+ed*(-C1+C5*ed-C6*delz*ee)
        +delz*(C2*ee+delz*(-C3*ec+delz*C4*ea)))/(ave*sqrt(ave));
}
```

```c
#include <math.h>
#include "nrutil.h"
#define ERRTOL 0.05
#define TINY 2.0e-13
#define BIG 9.0e11
#define C1 (3.0/14.0)
#define C2 (1.0/3.0)
#define C3 (3.0/22.0)
```

```
#define C4 (3.0/26.0)
#define C5 (0.75 * C3)
#define C6 (1.5 * C4)
#define C7 (0.5 * C2)
#define C8 (C3+C3)

float rj(float x, float y, float z, float p)
```

计算第三类卡尔森椭圆积分，$R_J(x,y,z,p)$。$x,y,z$ 必须为非负，至多一个为零，$p$ 必须不为零。若 $p<0$，返回柯西主值。TINY 至少须为机器下溢极限的立方根的2倍。BIG 不能超过机器上溢极限的立方根的五分之一。

```
{
    float rc(float x, float y);
    float rf(float x, float y, float z);
    float a,alamb,alpha,ans,ave,b,beta,delp,delx,dely,delz,ea,eb,ec,
        ed,ee,fac,pt,rcx,rho,sqrtx,sqrty,sqrtz,sum,tau,xt,yt,zt;

    if (FMIN(FMIN(x,y),z) < 0.0 || FMIN(FMIN(x+y,x+z),FMIN(y+z,fabs(p))) < TINY
        || FMAX(FMAX(x,y),FMAX(z,fabs(p))) > BIG)
            nrerror("invalid arguments in rj");
    sum=0.0;
    fac=1.0;
    if (p > 0.0) {
        xt=x;
        yt=y;
        zt=z;
        pt=p;
    } else {
        xt=FMIN(FMIN(x,y),z);
        zt=FMAX(FMAX(x,y),z);
        yt=x+y+z-xt-zt;
        a=1.0/(yt-p);
        b=a*(zt-yt)*(yt-xt);
        pt=yt+b;
        rho=xt*zt/yt;
        tau=p*pt/yt;
        rcx=rc(rho,tau);
    }
    do {
        sqrtx=sqrt(xt);
        sqrty=sqrt(yt);
        sqrtz=sqrt(zt);
        alamb=sqrtx*(sqrty+sqrtz)+sqrty*sqrtz;
        alpha=SQR(pt*(sqrtx+sqrty+sqrtz)+sqrtx*sqrty*sqrtz);
        beta=pt*SQR(pt+alamb);
        sum += fac*rc(alpha,beta);
        fac=0.25*fac;
        xt=0.25*(xt+alamb);
        yt=0.25*(yt+alamb);
        zt=0.25*(zt+alamb);
        pt=0.25*(pt+alamb);
        ave=0.2*(xt+yt+zt+pt+pt);
        delx=(ave-xt)/ave;
        dely=(ave-yt)/ave;
        delz=(ave-zt)/ave;
        delp=(ave-pt)/ave;
    } while (FMAX(FMAX(fabs(delx),fabs(dely)),
        FMAX(fabs(delz),fabs(delp))) > ERRTOL);
    ea=delx*(dely+delz)+dely*delz;
    eb=delx*dely*delz;
    ec=delp*delp;
    ed=ea-3.0*ec;
    ee=eb+2.0*delp*(ea-ec);
    ans=3.0*sum+fac*(1.0+ed*(-C1+C5*ed-C6*ee)+eb*(C7+delp*(-C8+delp*C4))
```

```
      +delp*ea*(C2-delp*C3)-C2*delp*ec)/(ave*sqrt(ave));
    if (p <= 0.0) ans=a*(b*ans+3.0*(rcx-rf(xt,yt,zt)));
    return ans;
}
```

```
#include <math.h>
#include "nrutil.h"
#define ERRTOL 0.04
#define TINY 1.69e-38
#define SQRTNY 1.3e-19
#define BIG 3.e37
#define TNBG (TINY * BIG)
#define COMP1 (2.236/SQRTNY)
#define COMP2 (TNBG * TNBG/25.0)
#define THIRD (1.0/3.0)
#define C1 0.3
#define C2 (1.0/7.0)
#define C3 0.375
#define C4 (9.0/22.0)

float rc(float x, float y)
```

计算卡尔森退化椭圆函数 $R_C(x,y)$。$x$ 必须非负，$y$ 必须非零。若 $y<0$，返回柯西主值。TINY 必须至少是机器下溢极限的五倍，BIG 至多为机器最大上溢极限的五分之一。

```
{
    float alamb,ave,s,w,xt,yt;
    if (x < 0.0 || y == 0.0 || (x+fabs(y)) < TINY || (x+fabs(y)) > BIG ||
        (y<-COMP1 && x > 0.0 && x < COMP2))
            nrerror("invalid arguments in rc");
    if (y > 0.0) {
        xt=x;
        yt=y;
        w=1.0;
    } else {
        xt=x-y;
        yt = -y;
        w=sqrt(x)/sqrt(xt);
    }
    do {
        alamb=2.0*sqrt(xt)*sqrt(yt)+yt;
        xt=0.25*(xt+alamb);
        yt=0.25*(yt+alamb);
        ave=THIRD*(xt+yt+yt);
        s=(yt-ave)/ave;
    } while (fabs(s) > ERRTOL);
    return w*(1.0+s*s*(C1+s*(C2+s*(C3+s*C4))))/sqrt(ave);
}
```

有时候可能希望以勒让德记号来表示答案。或者也许是，已有了以这种方法表示的结果，而希望用上面给出的程序计算其值。这种来回转换并不难。**第一类勒让德椭圆积分**定义为

$$F(\varphi,k) \equiv \int_0^{\varphi} \frac{d\theta}{\sqrt{1-k^2\sin^2\theta}} \tag{6.11.17}$$

**第一类完全椭圆积分**由下式给定

$$K(k) \equiv F(\pi/2,k) \tag{6.11.18}$$

根据 $R_F$ 有

$$F(\varphi,k)=\sin\varphi\, R_F(\cos^2\varphi, 1-k^2\sin^2\varphi, 1)$$
$$K(k)=R_F(0,1-k^2,1) \tag{6.11.19}$$

**第二类勒让德椭圆积分及第二类完全椭圆积分**由下式给定

$$\begin{aligned}E(\varphi,k)&\equiv\int_0^{\varphi}\sqrt{1-k^2\sin^2\theta}\,d\theta\\&=\sin\varphi\, R_F(\cos^2\varphi,1-k^2\sin^2\varphi,1)\\&\quad-\frac{1}{3}k^2\sin^3\varphi\, R_D(\cos^2\varphi,1-k^2\sin^2\varphi,1)\end{aligned}\tag{6.11.20}$$

$$E(k)\equiv E(\pi/2,k)=R_F(0,1-k^2,1)-\frac{1}{3}k^2R_D(0,1-k^2,1)$$

最后给出**第三类勒让德椭圆积分**

$$\begin{aligned}\Pi(\varphi,n,k)&\equiv\int_0^{\varphi}\frac{d\theta}{(1+n\sin^2\theta)\sqrt{1-k^2\sin^2\theta}}\\&=\sin\varphi R_F(\cos^2\varphi,1-k^2\sin^2\varphi,1)\\&\quad-\frac{1}{3}n\sin^3\varphi\, R_J(\cos^2\varphi,1-k^2\sin^2\varphi,1,1+n\sin^2\varphi)\end{aligned}\tag{6.11.21}$$

(注意,这里 $n$ 的符号习惯与 Abramowtz 和 Stegun[12]的记号相反,他们记号中 $\sin\alpha$ 即为这里的 $k$。)

```
#include <math.h>
#include "nrutil.h"

float ellf(float phi, float ak)    用卡尔森函数 R_F 求勒让德第一类椭圆积分 F(φ,k)
{
    float rf(float x, float y, float z);
    float s;

    s=sin(phi);
    return s * rf(SQR(cos(phi)),(1.0-s * ak) * (1.0+s * ak),1.0);
}

#include <math.h>
#include "nrutil.h"

float elle(float phi, float ak)    用卡尔森函数 R_D 和 R_F 求第二类勒让德椭圆积分 E(φ,k)
{
    float rd(float x, float y, float z);
    float rf(float x, float y, float z);
    float cc,q,s;

    s=sin(phi);
    cc=SQR(cos(phi));
    q=(1.0-s * ak) * (1.0+s * ak);
    return s * (rf(cc,p,1.0)-(SQR(s * ak)) * rd(cc,q,1.0)/3.0);
}

#include <math.h>
#include "nrutil.h"

float ellpi(float phi, float en, float ak)
    用卡尔森函数 R_J 和 R_F 求第三类勒让德椭圆积分 Π(φ,n,k)(注意 n 的记号习惯与 Abramowity 和 Stegun 的相反)
```

```
{
    float rf(float x, float y, float z);
    float rj(float x, float y, float z, float p);
    float cc,enss,q,s;

    s=sin(phi);
    enss=en*s*s;
    cc=SQR(cos(phi));
    q=(1.0-s*ak)*(1.0+s*ak);
    return s*(rf(cc,q,1.0)-enss*rj(cc,q,1.0+enss)/3.0);
}
```

卡尔森函数是−1/2阶和−3/2阶的齐次函数，所以

$$R_F(\lambda x,\lambda y,\lambda z)=\lambda^{-1/2}R_F(x,y,z)$$
$$R_J(\lambda x,\lambda y,\lambda z,\lambda p)=\lambda^{-3/2}R_J(x,y,z,p) \tag{6.11.22}$$

这样，按勒让德记号表示Carlson函数，就将变量置换成升阶形式，利用齐次性将第三个变量映射到1，然后应用式(6.11.19)~(6.11.21)。

### 6.11.1 雅可比椭圆函数

定义雅可比(Jaccobi)椭圆函数 $sn$ 时，不是考察椭圆积分

$$u(y,k)\equiv u=F(\varphi,k) \tag{6.11.23}$$

而是考察其反函数

$$y=\sin\varphi=\mathrm{sn}(u,k) \tag{6.11.24}$$

等价地

$$u=\int_0^{sn}\frac{dy}{\sqrt{(1-y^2)(1-k^2y^2)}} \tag{6.11.25}$$

$k=0$时 sn 即为正弦函数。函数 cn、dn 由以下关系式定义：

$$\mathrm{sn}^2+\mathrm{cn}^2=1,\quad k^2\mathrm{sn}^2+\mathrm{dn}^2=1 \tag{6.11.26}$$

下面给的程序实际上取输入参数 $m_c\equiv k_c^2=1-k^2$，同时计算出所有三个函数 sn,cn,dn，因为计算三个并不比计算其中任何一个更困难。详细的方法见[8]。

```
#include <math.h>
#define CA 0.0003                CA 平方的精度

void sncndn(float uu, float emmc, float *sn, float *cn, float *dn)
    返回雅可比椭圆函数 sn(u,kc),cn(u,kc),dn(u,kc)，其中 uu=u,emmc=kc²。
{
    float a,b,c,d,emc,u;
    float em[14],en[14];
    int i,ii,l,bo;

    emc=emmc;
    u=uu;
    if (emc) {
        bo=(emc < 0.0);
        if (bo) {
            d=1.0-emc;
            emc /= -1.0/d;
            u *= (d=sqrt(d));
        }
        a=1.0;
        *dn=1.0;
```

```
        for (i=1;i<=13;i++) {
            l=i;
            em[i]=a;
            en[i]=(emc=sqrt(emc));
            c=0.5*(a+emc);
            if (fabs(a-emc) <= CA*a) break;
            emc *= a;
            a=c;
        }
        u *= c;
        *sn=sin(u);
        *cn=cos(u);
        if (*sn) {
            a=(*cn)/(*sn);
            c *= a;
            for (ii=l;ii>=1;ii--) {
                b=em[ii];
                a *= c;
                c *= (*dn);
                *dn=(en[ii]+a)/(b+a);
                a=c/b;
            }
            a=1.0/sqrt(c*c+1.0);
            *sn=(*sn >= 0.0 ? a : -a);
            *cn=c*(*sn);
        }
        if (bo) {
            a=(*dn);
            *dn=(*cn);
            *cn=a;
            *sn /= d;
        }
    } else {
        *cn=1.0/cosh(u);
        *dn=(*cn);
        *sn=tanh(u);
    }
}
```

参考文献及进一步读物：


Erdelyi, A., Magnus, W., Oberhettinger, F., and Tricomi, F.G. 1953, *Higher Transcendentol Functions*, vol. II, (New York:McGraw-Hill). [1]

Gradshteyn, I. S., and Ryzhik, I. W. 1980, *Table of integrals, Series, and Products*(New York Academic Pres). [2]

Carlson, B.C. 1977, *SIAM Journal on Mathematical Analysis*, vol. 8, pp. 231～242. [3]

Carlson, B.C. 1987, *Mathematics of Computation*, vol. 49, pp. 595～606 [4]; 1988, *op. cit*, vol 51, pp. 267～280[5]; 1989, *op. cit.*, vol. 53, pp. 327 ～333 [6]; 1991, *op. cit.*, vol. 56, pp. 267～280. [7]

Bulirsch, R. 1965, *Numerischem Mathematik*, vol. 7, pp. 78～90; 1965, *op. cit.*, vol. 7, pp. 353～354; 1969, *op. cit.*, vol. 13, pp. 305～315. [8]

Carlson, B.C. 1979, *Numerische Mathematick*, vol. 33, pp. 1～16. [9]

Carlson, B.C., and Notis, E.M. 1981, *ACM Transactions on Mathematical Software*, vol. 7, pp. 398～403. [10]

Carlson, B.C. 1978, *SIAM Journal on Mathematical Analysis*, vol. 9, p. 524～528. [11]

Abramowitz, M., and Stegun, I. A 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, vol. 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications,

New, York), Chapter 17. [12]

## 6.12 超几何函数

正如第 5.14 节讨论过的那样，要找到一个计算复超几何函数 ${}_2F_1(a,b,c;z)$ 的快速的通用程序，如果不是不可能的，也是非常困难的。该函数定义为下面超几何级数的解析延拓：

$$
{}_2F_1(a,b,c;z) = 1 + \frac{ab}{c}\frac{z}{1!} + \frac{a(a+1)b(b+1)}{c(c+1)}\frac{z^2}{2!} + \cdots
+ \frac{a(a+1)\ldots(a+j-1)b(b+1)\ldots(b+j-1)}{c(c+1)\ldots(c+j-1)}\frac{z^j}{j!} + \cdots \tag{6.12.1}
$$

此级数仅在单位圆 $|z|<1$ 内收敛（见[1]），但我们感兴趣的不只是该区域。

第 5.14 节讨论过用在复平面上，直接线积分求此函数值的方法，这里仅列出得到的结果程序。

函数 **hypgeo** 的实现，正如程序中注解描述的那样，是很直接显然的。16 章中对微分的程序 **odeint** 很好用，甚至不必完全搞清楚；使用 **odeint** 需要一个为零的全程变量、一次函数的调用、以及对求导程序 **hypdrv** 预先设定格式。

当然，当 $z$ 值太靠近奇异点1时，函数 **hypgeo** 会失效。（若想靠近此奇异点，或在∞处的奇异点，可用[1]中第 15.3 节"线性转换公式"）。若远离 $z=1$，且 $a,b,c$ 值适当，则积分往往只需要令人吃惊的很少几步即可完成。一般为 6 步。

```
#include <math.h>
#include "complex.h"
#include "nrutil.h"
#define EPS 1.0e-6                       精度参数

fcomplex aa,bb,cc,z0,dz;                 传递给 hypdrv

int kmax,kount;                          由 odeit 使用
float *xp,**yp,dxsav;

fcomplex hypgeo(fcomplex a, fcomplex b, fcomplex c, fcomplex z)
    通过直接对复平面上超几何方程积分，对复数 a,b,c,z 计算复超几何函数 2F1，分支切换点取在沿实轴，Re z>1
处
{
    void bsstep(float y[], float dydx[], int nv, float *xx, float htry,
        float eps, float yscal[], float *hdid, float *hnext,
        void (*derivs)(float, float [], float []));
    void hypdrv(float s, float yy[], float dyyds[]);
    void hypser(fcomplex a, fcomplex b, fcomplex c, fcomplex z,
        fcomplex *series, fcomplex *deriv);
    void odeint(float ystart[], int nvar, float x1, float x2,
        float eps, float h1, float hmin, int *nok, int *nbad,
        void (*derivs)(float, float [], float []),
        void (*rkqs)(float [], float [], int, float *, float, float,
        float [], float *, float *, void (*)(float, float [], float [])));
    int nbad,nok;
    fcomplex ans,y[3];
    float *yy;

    kmax=0;
    if (z.r*z.r+z.i*z.i <= 0.25) {                 使用级数 ...
```

```
        hypser(a,b,c,z,&ans,&y[2]);
        return ans;
    }
    else if (z.r < 0.0) z0=Complex(-0.5,0.0);      ...或为线积分选择起点
    else if (z.r <= 1.0) z0=Complex(0.5,0.0);
    else z0=Complex(0.0,z.i >= 0.0 ? 0.5 : -0.5);
    aa=a;                                           通过对odeint前面的全程变量赋值,
    bb=b;                                              给hypdrv传递参数
    cc=c;
    dz=Csub(z,z0);
    hypser(aa,bb,cc,z0,&y[1],&y[2]);                求起始函数和导数
    yy=vector(1,4);
    yy[1]=y[1].r;
    yy[2]=y[1].i;
    yy[3]=y[2].r;
    yy[4]=y[2].i;
    odeint(yy,4,0.0,1.0,EPS,0.1,0.0001,&nok,&nbad,hypdrv,bsstep); odeint的参数见16.2节
    y[1]=Complex(yy[1],yy[2]);
    free_vector(yy,1,4);
    return y[1];
}
```

```
#include "complex.h"
#define ONE Complex(1.0,0.0)

void hypser(fcomplex a, fcomplex b, fcomplex c, fcomplex z, fcomplex *series, fcomplex *deriv)
```

返回超几何级数 $_2F_1$ 及其导数,迭代到机器精度,$|z|\leqslant 1/2$ 时收敛很快.

```
{
    void nrerror(char error_text[]);
    int n;
    fcomplex aa,bb,cc,fac,temp;

    deriv->r=0.0;
    deriv->i=0.0;
    fac=Complex(1.0,0.0);
    temp=fac;
    aa=a;
    bb=b;
    cc=c;
    for (n=1;n<=1000;n++) {
        fac=Cmul(fac,Cmul(aa,Cdiv(bb,cc)));
        deriv->r+=fac.r;
        deriv->i+=fac.i;
        fac=Cmul(fac,RCmul(1.0/n,z));
        *series=Cadd(temp,fac);
        if (series->r == temp.r && series->i == temp.i) return;
        temp= *series;
        aa=Cadd(aa,ONE);
        bb=Cadd(bb,ONE);
        cc=Cadd(cc,ONE);

    }
    nrerror("convergence failure in hypser");
}
```

```
#include "complex.h"
#define ONE Complex(1.0,0.0)

extern fcomplex aa,bb,cc,z0,dz;                     在 hypgeo 中定义
```

```
void hypdrv(float s, float yy[], float dyyds[])      计算超几何方程的导数,见文中式(5.14.4)
{
    fcomplex z,y[3],dyds[3];

    y[1]=Complex(yy[1],yy[2]);
    y[2]=Complex(yy[3],y[4]);
    z=Cadd(z0,RCmul(s,dz));
    dyds[1]=Cmul(y[2],dz);
    dyds[2]=Cmul(Csub(Cmul(Cmul(aa,bb),y[1]),Cmul(Csub(cc,
        Cmul(Cadd(Cadd(aa,bb), ONE), z)), y[2])),
        Cdiv(dz,Cmul(z,Csub(ONE,z))));
    dyyds[1]=dyds[1].r;
    dyyds[2]=dyds[1].i;
    dyyds[3]=dyds[2].r;
    dyyds[4]=dyds[2].i;
}
```

**参考文献和进一步读物:**


Abramowita, M., and Stegun, I. A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Seriesm, vol. 55. [1]

# 第七章 随机数

## 7.0 引言

利用计算机，这种人类所设计的各种机器中最精确、最能作出确切判断的机器，来产生"随机数"，这看上去似乎有些自相矛盾。甚至在概念上是讲不通的。任何程序必将产生出完全可以预计的结果，因而不是真正的"随机"数。

然而，计算机的实用"随机数生成程序"却得到广泛的使用。因此这个自相矛盾的问题还是留给哲学家们去深刻的解释吧！（参见 KNUTH 的第3.5节的讨论和文献）。人们有时把计算机生成的序列称作**伪随机**，而将**随机**一词专用来指内在的随机物理过程所产生的输出，例如，放在放射性元素样品旁，Geiger 计数器的卡嗒声之间所消逝的时间。在此我们不准备对伪随机和随机做严格的区分。

随机性的定义虽不精确，但在计算机生成序列的叙述中指出，产生随机序列的程序应该不同于使用其输出的计算机程序，并且（在所有可测量的方面）与后一程序在统计上是不相关的。换言之，任何两个不同的随机数生成程序，当它们与用户的特定应用程序相结合时，都应该产生统计上相同的结果。如果不是这样，（按用户的观点）至少其中一个不是好的随机数生成程序。

上述解释看起来是循环的，因为它将一个程序与另一个程序相互作了比较。然而，确实存在一批随机数生成程序，这些程序对很广泛的一类应用程序都能满足定义。根据经验还发现，用物理过程产生的随机数来检验，能获得统计上相同的结果。正因为存在这样的生成程序，所以，我们可以把下定义的问题留给哲学家们。

那么，从实用的观点来看，随机性是使用者（或应用程序员）自己认可的事。对某一应用问题来说已足够随机的生成程序，但未必对另一个应用问题也具有足够的随机性。当然，人们不会因那些不相适应的应用程序而完全不知所措：可以用一些现有的统计测试表来进行测试，这类表中某些表是实用的，某些表只是因为历史的原因而奉若神明。但总的来说，它们在测试相关性方面都能做得十分出色，如同通过应用程序（用户的应用程序）进行测试一样。好的随机数生成程序应该能够通过所有这些测试；或者使用者至少能够知道它们中哪一个会失败，所以使用者应该有能力判断这些生成程序是否与所探讨的情形有关。

关于这一主题的参考资料，首先值得阅读的是 Knuth[1] 的书。至于数值方法的标准书籍[2-4]中，只有很少几本是将随机数作为主题来论述的。

参考文献和进一步的读物：

Knuth, D. E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), Chapter 3, especially § 3.5. [1]

Bratley, P., Fox, B. L., and Schrage, E. L. 1983, *A Guide to Simulation* (New York: Springer-Verlag).

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ:Prentice-Hall), Chapter 11. [3]

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ:Prentice-Hall), Chapter 10. [4]

## 7.1 一致偏离

一致偏离就是在一特定范围内(典型为0至1之间)的随机数。在这个范围内,任一个数出现的可能性是等同的,换言之,一致偏离大概就是人们所认为的"随机数"。不过,我们总要将一致偏离与其他种类的"随机数"区分开来,例如,将它与给定的均值和标准差的正态分布所产生的数区分开来。正如我们将在后面各节看到的那样,这些其它种类的偏离,几乎总是要通过对一个或多个一致偏离进行适当的运算而生成的。所以,这一节的主题是:随机一致偏离,它是建立任何一种随机模型或蒙特卡罗(Monto Caolo)计算工作的基本构造模块。

### 7.1.1 系统提供的随机数生成程序

多数 $c$ 语言内部蕴含有可以初始化并能生成"随机数"的 ANSI 库函数。典型程序为:

```
#include <stdlib.h>
#define RAND_MAX ...

void srand(unsigned seed);
int rand(void);
```

当第一次调用 **srand** (**seed**)之前,应该给 **seed** 赋一个任意的初始值。每个初始值都会典型地返回不同的随机序列,或者至少是某个极长序列的不同的子序列。不过在函数 **seed** 中相同的初始值总是返回相同的随机序列。

采用连续调用 **rand**(),可以在序列中得到连续的随机数。这个函数返回一个整数,它通常处于0至最大的可表示的正整数之间(含最大正值整数)。通常,作为 ANSI C 标准,这一最大值可用 RAND_MAX 来表示;有时则需要用户自己找出来。如果需在0.0(含0.0)和1.0(不含1.0)之间得到一个随机浮点数,可用以下表示方法:

```
x=rand()/(RAND_MAX+1.0)
```

在本章中,这一节最重要的内容是,对系统提供的 **rand**()产生很大很大的怀疑,因为它同我们刚刚叙述的内容很类似。由于不合理的函数 **rand**()使一些科学论文的结果值得怀疑,如果将这些科学论文从图书馆的书架上全部取走的话,那么在每个书架上就会留出一拳之宽空隙。因为系统提供的 **rand**()几乎总是**线性同余生成程序**,它是按以下的递推关系生成整数序列 $I_1, I_2, I_3, \cdots$,每个数都在0和 $m-1$之间(例如,RAND_MAX)。

$$I_{j+1} = aI_j + c \qquad (\mathrm{mod}\ m) \tag{7.1.1}$$

这里 $m$ 称作**模**,$a$ 和 $c$ 是正整数,分别称作**乘子**和**增量**。递推式(7.1.1)一直反复进行,其周期显然不超过 $m$。如果恰当的选取 $m$、$a$ 和 $c$,则将获得最大周期为长度 $m$。在这种情况下,0与 $m-1$之间的所有可能的整数都在某点出现,因而任何初始"种子"$I_0$的选择都与其他选择一样:序列恰好从该点产生。

尽管这种一般的结构已足以能提供相当不错的随机数,但它的实现方法也很多。如果不

是这样，ANSI C 语言库是相当有缺陷的；但其中一定数量的实施方法只能“拙劣不堪地工作”。这个责任将由 ANSI C 委员会和他们的实施者来承担。典型的问题如下：

首先，由 ANSI C 标准规定 **rand**()返回一个整数型的数值，这个整数在许多机器中只有两个字节的长度—使RAND_MAX 通常不是很大。ANSI C 标准要求 RAND_MAX 最多只是32767。在许多情况下，这可能是灾难性的。对蒙特卡罗积分(第7.6节)和(第7.8节)来说，大概需要对$10^6$个不同点进行计算求值。于是，根本不相同的每一次计算求值，实际上却是对同样的32767个点进行30次运算！因此，必须明确地丢弃一些库内只具有两个字节返回值的随机数程序。

其次，ANSI 委员会发表的理论说明中包含了以下有害的一段话：“委员会规定，一种实施方法所提供的 **rand** 函数，它能在该实施过程中产生最好的随机序列，那么，这种实施方法是允许的。所以，没有规定标准的算法。这就是承认在不同的实施方法中，函数能产生相同的伪随机序列，正如已发表的实例…”，这个“例子”是

```
unsigned long next=1;

int rand (void)   /* NOT RECOMMENDED (see text) */
{
    next=next * 1103515245+12345;
    return (unsigned int)(next/65536) % 32768;
}

void srand(unsigned int seed)
{
    next=seed;
}
```

相当于式(7.1.1)中取 $a=1103515245$，$c=12345$，$m=2^{32}$(因此，算法在无符号长精度数据类型下计算，能保证返回正确的低阶位)。在该例中 $a$ 和 $c$ 虽并非是最好的选择，但总的来说，对它们的选择不是特别复杂。然而，真正弄巧成拙的是，执行者们把委员会的上述陈述作为依据，试着在发表例子上作“改进”。例如，在一台普通的 32 位 PC 兼容机上，装上一个与上述相符的长型类的生成程序，但它交换了返回值的高 16 比特位和低 16 比特位。有人认为，这种流行的额外的随机数实际上在破坏程序。当错误混合在一起时，就会在线性同余生成程序中，形成原始的根本错误。这就是我们现在正要讨论的问题。

线性同余的方法具有速度快、每次调用只需少量运算等优点，因此几乎是通用的。但它的缺点是在连续调用时，不能避免序列的相关性。如果同时使用 $k$ 个随机数在 $k$ 维空间中描点作图(每个坐标都在0和1之间)，这些点不会出现“充满”$k$ 维空间的趋势，相反地将位于 $k-1$维“平面”上，这样的平面最多大约有 $m^{1/k}$个。并且如果常数 $m$，$a$ 和 $c$ 不是选得很仔细的话，则平面的个数还要少许多。如果数 $m$ 为 32768，那么在三维空间中，三重点位于的平面数将不超过 32768($2^{15}$) 的立方根，即 32。既使 $m$ 接近机器所表示的最大整数，例如～$2^{32}$，那么三重点所位于的平面数，通常不超过 $2^{32}$的立方根，约为 1600。那么，用户不如将注意力只集中在总体积中一小部分范围内发生的物理过程上，所以平面的离散性非常明显。

更糟的是，读者可能正在使用一个程序，它选择的 $m$，$a$ 和 $c$ 往往弄巧成拙。一个很差劲的这种程序如 RANDU，其 $a=65539$ 和 $2^{31}$，在 IBM 计算机上流传了许多年，并被广泛复制在其他系统上[1]。我们中的一位检测产生了只具有11个平面的一个“随机”图。他所在的计算

机中心程序设计顾问告诉他，这是误用了随机数生成程序因此得出这种结论："我们保证每一个数单独来说都是随机的，但是并不保证一个以上的数也是随机的"。

$k$ 维空间的相关性并非是线性同余生成程序的唯一缺陷。这种生成程序的低阶位(最小有效位)常常比它们的高阶位随机性差得多。如果想在 1 和 10 之间生成一个随机整数，应该这样做：

```
j=1+(int) (10.0*rand()/(RAND_MAX+1.0));
```

而决不应该用：

```
j=1+(rand() % 10);
```

(它使用了低阶位)。同样，决不能将"**rand**()"数分成几个想像的随机段，而是应该对每段采取不同的调用。

### 7.1.2 可移植的随机数生成程序

Park 和 Miller[1]研究出大量的随机数生成程序。这些程序迄今为止已被使用了近30多年。随着严格的理论检验，他们指出了一些正在被广泛使用的不适当的程序实例。

这里有一个很好的经过理论与实际检验的论证，这就是简单的乘法同余算法：

$$I_{j+1} = aI_j \qquad (\text{mod } m) \tag{7.1.2}$$

如果乘数 $a$ 和模 $m$ 选择得仔细精确，它可以和 $c\neq0$(式(7.1.1))的线性同余生成程序一样好。Park 和 Miller 提出了一个基于以下选择的"最低标准程序"：

$$\begin{aligned} a &= 7^5 = 16807 \\ m &= 2^{31} - 1 = 2147483647 \end{aligned} \tag{7.1.3}$$

这个程序在1969年首先由 Lewis，Goodman 和 Miller 提出，这个程序在以后的几年里，通过了所有新型理论的检验，并且(也许更重要)积累了大量成功的应用经验。Park 和 Miller 从未自称这一程序是"完美"的(在下面我们可看到它并非完美)，但是与其他程序相比较，经鉴定这个最低标准程序是最好的。

在高级语言中，直接实施式(7.1.2)和(7.1.3)是不可能的，因为 $a$ 和 $m-1$的乘积超过了 32 位整数的最大值。在汇编语言的执行过程中，使用 64 位乘积寄存器是很简单的，但用于机器与机器之间的数据传递就很不方便。当两个 32 位整数之积以一个 32 位的常数为模时，不用借助其他大于 32 位(包括符号位)的辅助寄存器，这个诀窍应归于 Schrage[2,3]。对此非常感兴趣的是：最低标准程序允许在所有机器中，用任何一种基本的程序语言来执行。

Schrage 的算法是基于 $m$ 的**近似因子分解**：

$$m = aq + r \quad 即 \quad q[m/a], \quad r = m \text{ mod } a \tag{7.1.4}$$

方括号表示整数部分。若 $r$ 很小，特别是 $r<q$ 且 $o<z<m-1$，则可以看出 $a(z \text{ omd } q)$ 和 $r[z/g]$都处在0，…，$m-1$范围内，且

$$az \text{ mod } m = \begin{cases} a(z \text{ mod } q) - r[z/q] & 若它 \geq 0, \\ a(z \text{ mod } q) - r[z/q] + m & 其它 \end{cases} \tag{7.1.5}$$

对式(7.1.3)中的常数，Schrage 的算法取 $q=127773$，$r=2836$。

这里有一个最低标准程序：

```
#define IA 16807
#define IM 2147483647
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define MASK 123459876

float ran0(long *idum)
    Park 和 Miller 的"最低"随机数生成程序。返回在0.0至1.0之间的一致随机偏离。设置 idum 为任意整数，(除 MASK
    外)用于初始赋值。为序列中连续偏离而连续调用时，idum 值不允许改变。
{
    long k;
    float ans;

    *idum ^= MASK;                           对于 idum，与 MASK 异或允许使用零或其他简单的位模式
    k=(*idum)/IQ;
    *idum=IA*(*idum-k*IQ)-IR*k;              用 Schrage 法计算 idum=(IA*idum)%IM，不会溢出
    if (*idum < 0) *idum += IM;
    ans=AM*(*idum);                          变换 idum 为浮点型结果
    *idum ^= MASK;                           返回前去屏蔽
    return ans;
}
```

**ran0**的周期为 $2^{31}-2\approx 2.1\times 10^9$。在(7.1.2)形式的特殊程序中，0 值是决不允许做为初始种子—它只能维持它本身—它决不可能出现任何非0的初始种子。然而经验证明，使用者总是设法用种子 idum=0 来调用随机数生成程序。原因是，**ran0** 在入口和出口处都用任意常数来执行它的异或。如果用户能指出前人的错误，就可以把用 ^ 操作的这两行消去。

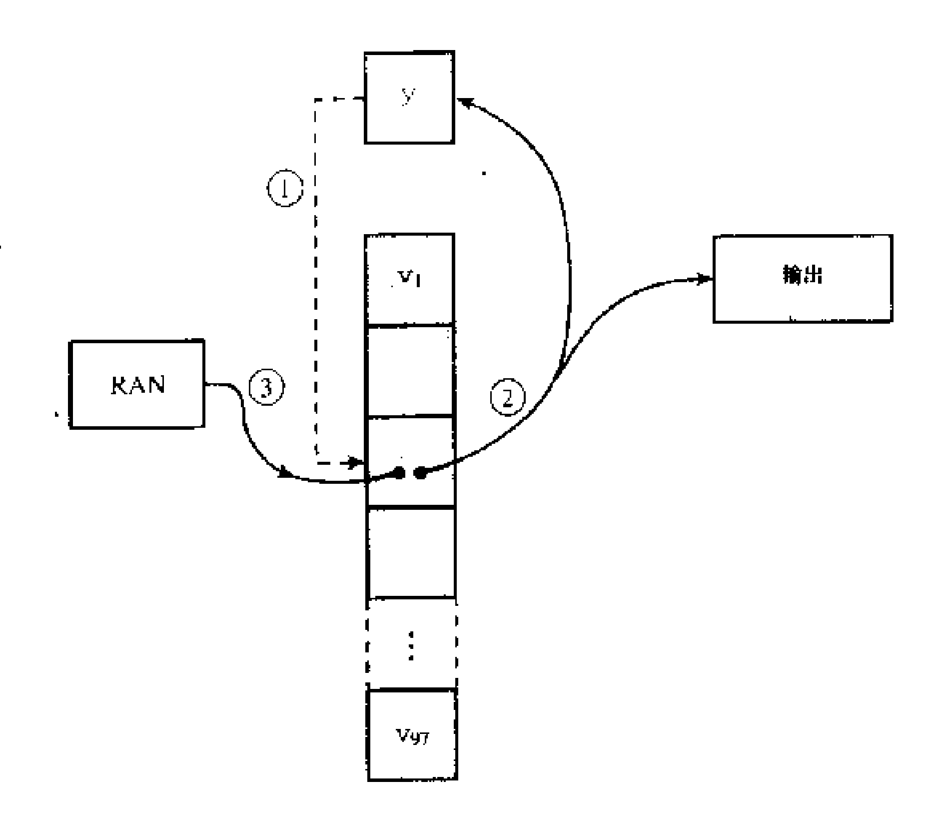
Park 和 Miller 讨论了另外两个乘子 $a$。它们都可用于同一个 $m=2^{31}-1$。它们是 $a=48271$(与 $q=44488$，$r=3399$)和 $a=69621$(与 $q=30845$，$r=23902$)。如果需要的话，这些值可在 **ran0**的程序中替换。它们比 Lewis 等所采用较长的试验值略有所强，除这些值之外，其他的值都不能用。

**ran0** 程序是一个最低标准，它的大多数应用是令人满意的。但我们并不把它作为随机数生成程序的最新成就来推荐。我们的理由是，这个最低标准是非常地简单。不难想象这种情况，在需使用连续的随机数处，它与生成程序的算法可能有抵触。例如，由于连续数与 $2\times 10^9$ 的模数只相差一个 $1.6\times 10^4$ 的倍数。因此，非常小的随机数之后趋于跟有一个小于平均值的数。例如，在 $10^6$ 的范围内，一旦有一个小于 $10^{-6}$ 的值返回，在这个值的后面总是跟有一个略小于0.0168 的数，这就很容易想到，如果把这些参数应用于特殊事件，就会导致错误的结果。

还有更难以捉摸的一系列相关性出现在 **ran0** 中。例如，如果连续的点$(I_i, I_{i+1})$属于二维平面，$i=1,2,...N$，那么当 $N$ 稍比$10^7$大一点，且远小于 $m-2$ 周期时，其结果使 $\chi^2$ 检验失败。因为低阶顺序相关性如此令人头痛，而且，现有的消除它们的方法又是非常简单，因此我们认为应谨慎地使用它。

下面的 **ran1**程序使用了最低标准产生随机值，但它为消除低阶顺序相关性而搅乱了输出。随机偏离要获取序列 $I_j$ 中的第 $j$ 个值，它不是取在第 $j$ 次调用中的输出值，而是取在调用平均 $j+32$ 次后的随机值。这种混乱的算法，往往就像 Bays 和 Darham 在 Knuth 中所

描述的那样，如图7.1.1中所阐述的。




图中加圆圈的数表示事件的顺序：在每次调用中，y中的随机数是用来选择数组v中的随机元素。该元素变成输出随机数，并且也就是下一个y，它在v中位置由最低标准程序重新填补

图7.1.1 用于ran1的混洗过程，它破坏了在最低标准随机数生成程序中的顺序相关性

```
#define IA 16807
#define IM 2147483647
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define NTAB 32
#define NDIV (1+(IM-1)/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)

float ran1(long *idum)
```

Parkit和Miller的“最低”随机数产生程序，包括Bays－Durham的混洗算法且加了保护。返回0.0～1.0之间的一致随机偏离，(除终点值外)。以idum的负整数作为初值调用，所以在连续偏离的序列中idum值不允许改变。RNMX近似为小于1的最大浮动值。

```
{
    int j;
    long k;
    static long iy=0;
    static long iv[NTAB];
    float temp;

    if (*idum <= 0 || !iy) {                初始赋值
        if (-(*idum) < 1) *idum=1;          可靠防止idum = 0
        else *idum = -(*idum);
```

```
        for (j=NTAB+7;j>=0;j--) {          装载混洗表(8以后递增)
            k=(*idum)/IQ;
            *idum=IA*(*idum-k*IQ)-IR*k;
            if (*idum < 0) *idum += IM;
            if (j < NTAB) iv[j] = *idum;
        }
        iy=iv[0];
    }
    k=(*idum)/IQ;                          非初值启动
    *idum=IA*(*idum-k*IQ)-IR*k;            用Schrage 的方法计算 idum=(IA*idum) %
    if (*idum < 0) *idum += IM;                IM不会溢出
    j=iy/NDIV;                             在 0..NTAB-1范围内
    iy=iv[j];                              预先保存值后输出并填充混洗表
    iv[j] = *idum;
    if ((temp=AM*iy) > RNMX) return RNMX;  使用者不要求终点值
    else return temp;
}
```

ran1程序通过了ran0没能通过的统计测试。事实上,我们并不知道ran1未能通过任何统计的测试,除了起始调用的次数变成周期 $m$ 的阶,例如大于$10^8 \approx m/20$。

对于某些场合需要更长的随机序列时,L'Ecuyer[6]给出了可将两个不同周期的序列组合,来形成一个新序列的方法,该序列的周期是这两个周期的最小公倍。基本原理是把两序列简单地相加,它们二者的模之一做为模,称它为 $m$。为避免中间值溢出整数值范围的好方法是相减而不是相加,然后再把常数 $m-1$加回去(如果结果$\leqslant 0$),这样就把数限制在所期望的区间0, . . . , $m-1$的范围内。

值得注意,这种受约束的减法可以从第一序列的每一个值得到所有的值0, . . . , $m-1$,但这并不是必须的。其实,考虑到不合理的极端情况,被减值只在1和10之间:所得的序列在随机性方面不亚于它自己的第一序列。实际上,第二个序列的值域基本上覆盖了第一个序列的值域,是必不可少的。L'Ecuyer 论述了两个程序的用法,$m_1=2147483563(a_1=40014, q_1=53668, r_1=12211)$和 $m_2=2147483399(a_2=40692, q_2=52774, r_2=3791)$。两个模数都略小于$2^{31}$,周期$m_1-1=2\times3\times7\times631\times81031$, $m_2-1=2\times19\times31\times1019\times1789$,只有一个公因子2,所以组合后的程序周期为$2.3\times10^{18}$。对于目前的计算机来说,超出这个周期是不可能的事。

两个程序的组合,在很大程度上破坏了顺序相关性。尽管如此,我们(在下面 **ran2**的程序中)将介绍相加混洗的执行过程。我们认为在它的浮点精度范围内,**ran2**提供了很好的随机数。这里对"很好"的定义是,我们将花1000美元奖励给第一个找出它错误的读者(用不寻常的方法列出 **ran2**在统计测试中的错误,机器浮点数表示的一般限制除外)。

```
#define IM1 2147483563
#define IM2 2147483399
#define AM (1.0/IM1-1)
#define IMM1 (IM1  1)
#define IA1 40014
#define IA2 40692
#define IQ1 53668
#define IQ2 52774
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1+IMM1/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)
```

float ran2(long *idum)

*L'Ecuyer* 的长周期(>2×10[18])随机数生成程序包括 Bays—Durham 的混洗算法而且加了保护。返回0.0~1.0之间的一致随机偏离,(除终点值外),调用 idum 的负整数为初值,在连续偏离序列中 idum 值不允许改变,RNMX 近似为小于1的最大浮动值。

```
{
    int j;
    long k;
    static long idum2=123456789;
    static long iy=0;
    static long iv[NTAB];
    float temp;

    if (*idum <=0){                                    初始赋值
        if(-(*idum)<1)    *idum=1;                     可靠防止 idum=0
        else  *idum= -(*idum);
        idum2=(*idum);
        for (j=NTAB+7;j>=0;j--)  {                     装载混洗表(8以后递增)
          k=(*idum)/IQ1;
          *idum=IA1*(*idum-k*IQ1)-k*IR1;
          if  (*idum<0)   *idum   +=IM1;
          if  (j<NTAB)  iv[j]=   *idum;
        }
        iy=iv[0]
    }
    k=(*idum)/IQ1;                                      非初始值启动
    *idum=IA1*(*idum-k*IQ1)-k*IR1;                      用 Schrage 的方法计算 idum=(IA1*idum)%IM1 不会
                                                        溢出
    if(*idum<0)*idum+=IM1;
    k=idum2/IQ2;
    idum2=IA2*(idum2-k*IQ2)-k*IR2;                      计算 idum2=(IA2*idum)%IM2不会溢出
    if  (idum2<0)  idum2 +=IM2;
    j=iy/NDIV;                                          在0...NTAB-1范围内
    iy=iv[j]-idum2;
    iv[j]= *idum;
    if (iy<1)   iy +=IMM1;                              这里 idum 已被混洗,idum 和 idum2组合产生输出
    if ((temp=AM*iy)   >RNMX)   return RNMX; 使用者不要求终点值
    else return temp;
}
```

L'Ecuyer[5]列出了一些附加的小程序,这些程序可以合并为一个大程序,包括可以执行16位整数计算。

最后,我们把 Knuth 建议[4]的可移植性程序提供给大家,我们已将它转换成为现在的程序,称作 **ran3**。这个 **ran3** 根本不是基于线性同余的方法,而是以一个相减方法为基础(见[5])。人们也许希望,如果这个程序有缺点,它应与前面的 **ran1** 的缺点具有截然不同的特征。如果用户总是对一个程序产生怀疑,那么一个好的办法是,在同一个应用程序中试用一下其他程序。**ran3** 具有一个漂亮的功能:如果所用的机器在整数算术方面(限于 16 位的整数)很差劲,你可以加注释,将 **mj**,**mk** 和 **ma**[]设为浮点数,并且定义 mbig 和 mseed 分别为 4000000 和 1618033 ,于是相应这个程序将全部变成浮点型程序。

```
#define MBIG 1000000000           根据 Knuth,任何很大的 MBIG 和任何较小
#define MSEED 161803398           的(但仍很大)MSEED 都可替换这些值
#define MZ 0
#define FAC (1.0/MBIG)

float ran3(long *idum)            返回0.0~1.0之间的一致随机偏离,设定 idum 为任意负值,作为序列的初始或
```

```
}                                   生成更新初始赋值
    static int inext,inextp;
    static long ma[56];             值56(其范围 ma[1...55])为一特殊值,故不应改动;参见 Knuth
    static int iff=0;
    long mj,mk;
    int i,ii,k;

    if (*idum<0 || iff==0){         初始赋值
        iff=1;
        mj=MSEED-(*idum < 0 ? -*idum : *idum);    使用种子 idum 和大数 MSEED 对 ma[55]
        mj %= MBIG;                               初始赋值
        ma[55]=mj;
        mk=1;
        for (i=1;i<=54;i++){        现在按稍微随机的顺序,并用不是特别随机的数,对表的其余部分初始赋值
            ii=(21*i) % 55;
            ma[ii]=mk;
            mk=mj-mk;
            if (mk < MZ) mk += MBIG;
            mj=ma[ii];
        }
        for (k=1;k<=4;k++)                        我们用"摇动生成程序的办法"来随机化它们
            for(i=1;i<=55;i++){
                ma[i] -= ma[1+(i+30) % 55];
                if (ma[i] < MZ) ma[i] += MBIG;
            }
        inext=0;                                  准备第一个生成数指标
        inextp=31;                                常数31为特殊数,参见 Knuth
        *idum=1;
    }
    if (++inext == 56) inext=1;                   增量 inext 和 inextp 在56至1间循环
    if (++inextp == 56) inextp=1;
    mj=ma[inext]-ma[inextp];                      相减而生成新随机数
    if (mj < MZ) mj += MBIG;                      应当确信它在范围之内
    ma[inext]=mj;                                 将其存贮
    return mj*FAC;                                并输出得到一致随机偏离
}
```

### 7.1.3 快速而略有缺陷的生成程序

人们经常需要将快速而略有缺陷的随机数生成程序嵌套在程序之中,也许只取一行或二行代码,仅仅稍微有一点随机的意思。例如,人们可能希望在并非始终相同顺序的实验中处理数据,所以,要求第一个输出要比可能出现的其他情形更"典型"。

对于这种应用,实际上真正需要的就是,"合理"地在式(7.1.1)中选择一组 $m$,$a$ 和 $c$,如果我们并不需要一个比$10^4$到$10^6$更长的周期,则我们可以维持$(m-1)a+c$ 的值足够小,使其小到足以在 Schrage 的复杂方法中能避免溢出。我们可以很容易地将

```
unsigned long jran,ia,ic,im;
float ran;
...
jran=(jran*ia+ic) % im;
ran=(float) jran / (float) im;
```

嵌入我们的程序。每当我们需要一个快速而略有缺陷的一致偏离时,或想得到在 jlo 和 jhi(包括两个端点在内)之间的一个整数时,则可以用

```
jran=(jran*ia+ic) % im;
j=jlo+((jhi-jlo+1)*jran)/im;
```

(在两种情况下,jran 曾取0和 $im-1$之间的任何种子值进行初始化。)

应当确认,当im很小时,它的第$k$次根,即为$k$维空间平面的个数,甚至会更小。所以一个快速却略有缺陷的程序将绝对不能用来选择$k>1$的$k$维空间的点。

有了以上的说明,对于"好"的常数的选择由下面的表7.1.1给出,这些常数(i)给出最大长度im的周期,并且更重要的是(ii)对2维、3维、4维、5维、6维的情形通过了Knuth的"谱检验"。增量ic是一个接近$\left(\frac{1}{2}-\frac{1}{6}\sqrt{3}\right)$im值的素数,实际上几乎与im互素的ic值都会通过这一检验,不过有些"经验知识"有利于这种选择(参见[4],48页)。

**表7.1.1 用于快速而略有缺陷的生成程序中的常数**

| 上溢出 | im | ia | ic |
|---|---|---|---|
| | 6075 | 106 | 1283 |
| $2^{20}$ | | | |
| | 7875 | 211 | 1663 |
| $2^{21}$ | | | |
| | 7875 | 421 | 1663 |
| $2^{22}$ | | | |
| | 6075 | 1366 | 1283 |
| | 6655 | 936 | 1399 |
| | 11979 | 430 | 2531 |
| $2^{23}$ | | | |
| | 14406 | 967 | 3041 |
| | 29282 | 419 | 6173 |
| | 53125 | 171 | 11213 |
| $2^{24}$ | | | |
| | 12960 | 1741 | 2731 |
| | 14000 | 1541 | 2957 |
| | 21870 | 1291 | 4621 |
| | 31104 | 625 | 6571 |
| | 139968 | 205 | 29573 |
| $2^{25}$ | | | |
| | 29282 | 1255 | 6173 |
| | 81000 | 421 | 17117 |
| | 134456 | 281 | 28411 |
| $2^{26}$ | | | |
| | 86436 | 1093 | 18257 |
| | 121500 | 1021 | 25673 |
| | 259200 | 421 | 54773 |
| $2^{27}$ | | | |
| | 117128 | 1277 | 24749 |
| | 121500 | 2041 | 25673 |
| | 312500 | 741 | 66037 |
| $2^{28}$ | | | |
| | 145800 | 3661 | 30809 |
| | 175000 | 2661 | 36979 |
| | 233280 | 1861 | 49297 |
| | 244944 | 1597 | 51749 |
| $2^{29}$ | | | |
| | 139968 | 3877 | 29573 |
| | 214326 | 3613 | 45289 |
| | 714025 | 1366 | 150889 |
| $2^{30}$ | | | |
| | 134456 | 8121 | 28411 |
| | 259200 | 7141 | 54773 |
| $2^{31}$ | | | |
| | 233280 | 9301 | 49297 |
| | 714025 | 4096 | 150889 |
| $2^{32}$ | | | |

## 7.1.4 更快的程序

在C语言中,若在32位长整型表示的机器中,要将两个无符号长整型整数相乘,返回值只是64位乘积中的低32位。如果我们选择$m=2^{32}$,则式(7.1.1)中的这个"模"是任意的,而且我们有简单的:

$$I_{j+1}=aI_j+c \tag{7.1.6}$$

Knuth建议$a=1664525$,做为$m$的一个适当的乘子,H. W. Lewis用$c=1013904223$,对$a$值进行了大量测试,它是一个接近$(\sqrt{5}-2)m$值的素数,这样结果的生成程序(我们将称它为**ranqd1**)是简单的:

```
unsigned long idum;
...
idum=1664525L*idum+1013904223L;
```

这个程序和任何32位线性同余程序一样有效,基本上满足许多应用的需要,而且只有一个单独的乘

法和加法，速度相当快。

为了检查机器是否具有所要求的整数性能，就要看它是否能产生下列32位值的序列(这里是以十六进制给出的)：00000000，3C6EF35F，47502932，D1CCF6E9，AAF95334，6252E503，9F2EC686，57FE6C2D，A3D95FA8，81FDBEE7，94F0AF1A，CBF633B1。

如果需要的是浮点值而不是32位整数，而且要避开浮点数除$2^{32}$，一个简单的方法是，把这个值掩蔽在某一个幂指数中，使其处于1和2之间，然后减1.0。这个合成的程序(我们称其为**ranqd2**)，看起来有点像如下程序：

```
    unsigned long idum,itemp;
    float rand;
#ifdef vax
    static unsigned long jflone=0x00004080;
    static unsigned long jflmsk=0xffff007f;
#else
    static unsigned long jflone=0x3f800000;
    static unsigned long jflmsk=0x007fffff;
#endif
    ...
    idum=1664525L*idum+1013904223L;
    itemp=jflone | (jflmsk & idum);
    rand=(*(float *)&itemp)-1.0;
```

十六进制常数3F800000和007FFFFF适合于用IEEE表示的计算机的32位浮点数(如IBM-PC机和大多数UNIX工作站)。对DEC-VAX机，正确的十六进制常数分别为00004084和FFFF007F。提醒注意，IEEE的掩蔽法会引起构造的浮点数超出整型数的23位低阶位，这不是理想的。(作者已尽最大努力，力求使这本书的全部内容独立于机器及兼容机，甚至与程序语言无关。当试验中，有时出现实在难以辨认的结果，的确无可反驳。这一小节可能略有误差，请原谅。)

### 7.1.5 相对的执行时间和建议

执行程序的时间必然取决于机器。然而下面的表7.1.2，表现出了典型机器中的相对的执行时间。在这一节中，讨论个别一致生成程序的相对执行时间，也包括第7.5节的**ran4**。在表7.1.2中，数值较小的表明程序执行速度较快。程序**ranqd1**和**ranqd2**可参考上面介绍的快速程序。

**表7.1.2**

| 程序 | 相对执行时间 |
|---|---|
| ran0 | ═ 1.0 |
| ran1 | ≈ 1.3 |
| ran2 | ≈ 2.0 |
| ran3 | ≈ 0.6 |
| ranqd1 | ≈ 0.10 |
| ranqd2 | ≈ 0.25 |
| ran4 | ≈ 4.0 |

总的来说，我们建议一般情况使用ran1。基于Park和Miller的附加混洗的标准生成程序是可移植的，但有关超出周期以外的问题，我们还没有研究。

如果在单个计算中要产生大于100000000的随机数(即，大于**ran1**周期的5%)，则我们建议用**ran2**程序，它的周期要长得多。

看上去Knuth的减法程序**ran3**对执行时间而言，在可移植程序中是佼佼者，遗憾的是，

这种方法不易掌握，而且不标准。我们一般把 **ran**3作为"第二位的建议"，当我们怀疑某个程序在相关计算上有缺点时，可用它来代替。

**ran**4程序是非常好的随机偏离程序，而且还具有其他一些优良性能，但它速度慢，参阅第7.5节。

最后，程序 **ranqd**1和 **ranqd**2的速度是非常快的，但它们依赖机器，而且最多与32位线性同余生成程序一样好。但以我们的观点，它们在很多场合下，并不是很满意的。我们只在速度受限制的特殊的情况下，使用它们。

参考资料和进一步阅读：


Park, S, K, and Miller, K. W. 1988. *Communications of the ACM*, vol. 31. pp. 1192～1201. [1]

Schrage, L. 1979, *ACM Transactions on Mathematical Software*, vol. 5, pp. 132～138. [2]

Bratley, P. Fox, B. L, and Schrage, E. L. 1983, *A Guide to Simulation* (New York: Springer-Verlag). [3]

Knuth, D. E. 1981. *Seminumerical Algorithms*, 2nded, vol. 2 of *The Artol Computer Programming* (Reading, MA: Addison-Wesley), §§3.2～3.3. [4]

Kahaner, D., Moler, C, and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs, NJ: Prentice Hall), Chapter 10. [5]

L'Ecuyer, P. 1988, *Communications of the ACM*, vol. 31. pp. 742～774. [6]


## 7.2 变换方法：指数偏离和正态偏离

前一节中，我们学习了如何用一致概率分布生成随机偏离的方法，所以在 x 和 dx 之间生成一个数的概率（记成 $p(x)dx$）可由下式给出：

$$p(x)dx = \begin{cases} dx & 0 < x < 1 \\ 0 & \text{其它} \end{cases} \tag{7.2.1}$$

概率分布 $p(x)$ 当然是归一化的，所以

$$\int_{-\infty}^{\infty} p(x)dx = 1 \tag{7.2.2}$$

现在，假设我们生成一个一致偏离 $x$，然后取 $x$ 的某个指定的函数 $y(x)$。$y$ 的概率分布由概率的基本变换定律所确定。简单地表示为：

$$|p(y)dy| = |p(x)dx| \tag{7.2.3}$$

或者

$$p(y) = p(x)\left|\frac{dx}{dy}\right| \tag{7.2.4}$$

### 7.2.1 指数偏离

例如：假设 $y(x) = -\ln(x)$，并且 $p(x)$ 由一致偏离的方程(7.2.1)给出。于是

$$p(y)dy = \left|\frac{dx}{dy}\right|dy = e^{-y}dy \tag{7.2.5}$$

它呈指数分布。在实际问题中，这种指数分布经常出现。通常，独立的泊松(Poisson)随机事件之间的等待时间的分布就是这种分布，例如，这事件可以是核放射性衰变。还可以很容易地（从7.2.3）看出，量 $y/\lambda$ 具有的概率分布为 $\lambda e^{-\lambda y}$。

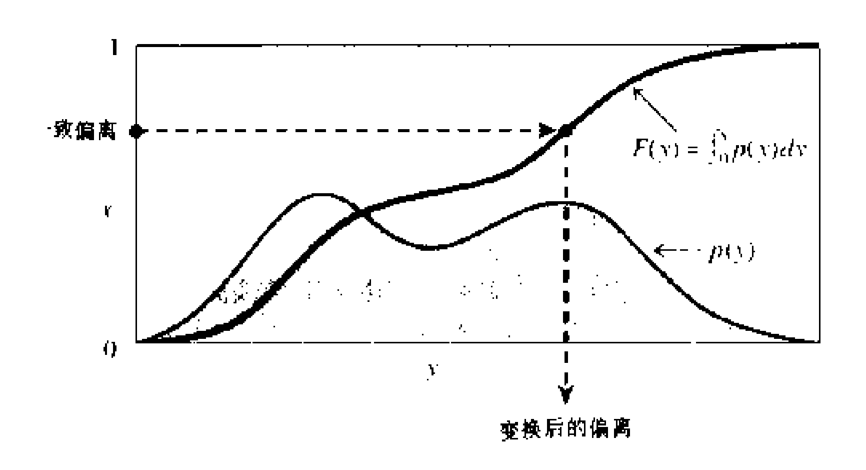
所以我们有：

```
#include <math.h>

float expdev(long *idum)        利用ran1(idum)作为一致偏离的来源,返回一个具单位均值、指数分布的正随机偏离
{
    float ran1(long *idum);
    float dum;

    do
        dum=ran1(idum);
    while (dum == 0.0);
    return -log(dum);
}
```

在使用上面的变换方法，生成 $y$ 的任意某个所期望的分布过程中，例如对某个积分等于 1 的正函数 $f$，生成具有 $p(y)=f(y)$ 的分布的过程中，让我们看看它包括一些什么内容（见图7.2.1）。




$p(y)$的不定积分必须是已知的并且是可逆的。一致偏离 $x$ 在0和1之间进行选择，它在给定的积分曲线上所对应的 $y$ 就是期望的偏离。

图7.2.1 由已知概率分布 $p(y)$ 生成随机偏离 $y$ 的变换方法

根据式(7.2.4)，我们需要求解微分方程

$$\frac{dx}{dy}=f(y) \tag{7.2.6}$$

这个方程的解正好是 $x=F(y)$，其中 $F(y)$ 是 $f(y)$ 的不定积分。所以，取 $f(y)$ 分布为一致偏离，它所期望的变换为

$$y(x)=F^{-1}(x) \tag{7.2.7}$$

其中 $F^{-1}$ 是 $F$ 的反函数。式(7.2.7)是否可行，取决于f(y)积分的反函数本身计算起来是否可行，无论从解析方面还从数值方面都是如此。计算起来有时候是可行的，有时则不可行。

顺便给式(7.2.7)一个直观的几何解释：既然 $F(y)$ 是 $y$ 的概率曲线下从0至 $y$ 的面积，式(7.2.7)就可按这样规定：选取一个一致随机的 $x$，然后寻找 $y$ 值，使从0到 $y$ 的概率面积对应于 $x$，即 $y$ 为所求之值。

### 7.2.2 正态(Gaussian)偏离

变换方法可以推广到高于一维的情形。如果 $x_1, x_2, \ldots$ 是具有联合概率分布 $p(x_1, x_2, \ldots)dx_1, dx_2, \ldots$ 的随机偏离,并且如果 $y_1, y_2, \ldots$ 每个都是所有 $x$ 的函数($y$ 的个数与 $x$ 的个数相同),则 $y$ 的联合的概率分布为:

$$p(y_1, y_2, \ldots)dy_1dy_2\ldots = p(x_1, x_2, \ldots)\left|\frac{\partial(x_1, x_2, \ldots)}{\partial(y_1, y_2, \ldots)}\right|dy_1dy_2\ldots \quad (7.2.8)$$

其中$|\partial()/\partial()|$为 $x$ 关于 $y$ 的雅可比行列式(或者是 $y$ 关于 $x$ 的雅可比行列式的倒数)。

运用式(7.2.8)的一个重要的例子是,生成具有正态分布的随机偏离的 Box—Muller 方法,

$$p(y)dy = \frac{1}{\sqrt{2\pi}}e^{-y^2/2}dy \quad (7.2.9)$$

考虑(0,1)内两个一致偏离 $x_1, x_2$ 与两个量 $y_1, y_2$ 之间的变换

$$\begin{aligned} y_1 &= \sqrt{-2\ln x_1}\cos 2\pi x_2 \\ y_2 &= \sqrt{-2\ln x_1}\sin 2\pi x_2 \end{aligned} \quad (7.2.10)$$

等价地,我们可以写成

$$\begin{aligned} x_1 &= \exp\left[-\frac{1}{2}(y_1^2 + y_2^2)\right] \\ x_2 &= \frac{1}{2\pi}\arctan\frac{y_2}{y_1} \end{aligned} \quad (7.2.11)$$

现在的雅可比行列式可以很容易地计算(请试试!)

$$\frac{\partial(x_1, x_2)}{\partial(y_1, y_2)} = \begin{vmatrix} \frac{\partial x_1}{\partial y_1} & \frac{\partial x_1}{\partial y_2} \\ \frac{\partial x_2}{\partial y_1} & \frac{\partial x_2}{\partial y_2} \end{vmatrix} = -\left[\frac{1}{\sqrt{2\pi}}e^{-y_1^2/2}\right]\left[\frac{1}{\sqrt{2\pi}}e^{-y_2^2/2}\right] \quad (7.2.12)$$

因为此式是单个 $y_2$的函数与单个 $y_1$函数的乘积,并且我们可以看出,每个 $y$ 都是服从正态分布式(7.2.9)的独立分布。

在应用式(7.2.10)时,一个进一步的决窍很有用。假设我们不在单位正方形中挑选一致偏离 $x_1$和 $x_2$,而代之用以原点为圆心的单位圆内挑选一个随机点,$v_1$和 $v_2$作为它的纵坐标和横坐标。于是,它们的平方和 $R^2 \equiv v_1^2 + v_2^2$是一个一致偏离,它可以用于 $x_1$。而由$(v_1, v_2)$定义的,关于 $v_1$的夹角可以起到随机角度 $2\pi x_2$的作用。这有什么好处?正是因为这点,现在式(7.2.10)中,余弦和正弦可以写作 $v_1/\sqrt{R^2}$和 $v_2/\sqrt{R^2}$,避免了对三角函数的调用!

于是有

```
#incude <math.h>

float gasdev(long * idum)          使用 ran1(idum)作为一致偏离的来源,返回一个具有零均值和
{                                  单位方差的正态分布偏离
    float ran1(long * idum)
    static int iset=0;
    static float gset;
    float fac,rsq,v1,v2;
```

```
    if(iset==0){                                  我们没有额外的偏离，所以
        do{
            v1=2.0*ran1(idum)-1.0;                由每个方向为-1扩充而成的正方形中选出
            v2=2.0*ran2(idum)-1.0;                两个一致数
            r=v1*v1+v2*v2;                        检查它们是否在单位圆内
        } while(rsq >= 1.0 || rsq == 0.0);        如果不在单位圆内，再试一次
        fac=sqrt(-2.0*log(rsq)/rsq);              现在作 Box Muller 变换，以得到两个正态偏离，返回一
        gset=v1*fac;                              个并为下次保存一个
        iset=1;                                   设置标志
        return v2*fac;
    } else {                                      我们有额外的偏离
        iset=0;                                   故清除标记
        return gset;                              并返回
    }
}
```

参阅 Devroye[1] 和 Bratley[2] 中许多其他算法。

参考文献和进一步读物：


Devroye, L. 1986, *Non-Uniform Random Variate Generation* (New York: Springer-Verlag), §9.1. [1]

Bratley, P., Fox, B. L., and Schrage, E. L. 1983, *A Guide to Simulation* (New York: Springer-Verlag). [2]


## 7.3 拒绝方法：Γ 偏离、泊松偏离、二项偏离

**拒绝方法**是生成随机偏离的强有力的一般技巧，这些随机偏离的分布函数 $p(x)dx$（在 $x$ 和 $x+dx$ 之间数值发生的概率）是已知的并可计算的。拒绝方法并不要求累积分布函数（$p(x)$的不定积分），是容易计算的，并且它比该函数的反函数小很多——这是前一节的变换方法所要求的。

拒绝方法是以简单的几何依据为基础的：

描绘一个想生成的概率分布 $p(x)$的图形，使得在任意 $x$ 范围内曲线下的面积，对应于在该范围内产生一个 $x$ 所期望的概率。如果我们有某种方法选择一个二维情形的随机点，使其在曲线下的面积具有一致概率，则该随机点 $x$ 值便是所期望的分布。

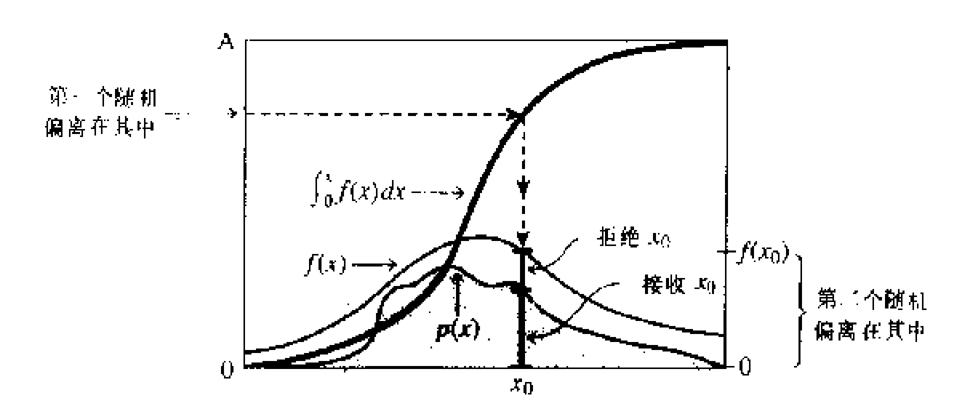
在这一图上画任一条曲线 $f(x)$，使之具有（并非无限）有限的面积，并且每一处都位于原来概率分布之上。（由概率的定义这始终是可能的，因为原来的曲线只围住单位面积。）我们将这个 $f(x)$称作**比较函数**。设想一下，若有在二维情形中选择一个随机点的某种方式，则该随机点在比较函数下面积是均匀的。每当该点位于原概率分布的面积以外，我们就拒绝它并选另一个随机点。每当该点位于原概率分布面积之内时，我们就接受它。很显然，接受点在被接受面积中是一致的，所以，这些 $x$ 值都有所期望的分布。而拒绝点的部分取决于比较函数的面积与概率分布函数的面积之比，而不取决于这两个函数之中任一个的具体形状。例如，面积小于 2 的比较函数将拒绝不超过一半的点。即使在某些 $x$ 值处，它非常差地逼近概率函数，如在某个 $x$ 为零的区域中，被拒绝的点仍保持有限。

剩下的只是，在比较函数 $f(x)$下如何选择二维的一致随机点的方法。变换方法（第7.2节）的变形能很好地解决这一问题：应当确信已选择了一个比较函数，它的不定积分是解析可知的，并且还是解析可逆的，以给出 $x$ 作为"在比较函数面积下直到 $x$ 左边"的函数。现在

选出位于 0 与 $A$ 之间的一个一致偏离，其中 $A$ 是 $f(x)$ 下总面积，并且用这个一致偏离得到一个对应的 x。然后，选择 0 与 $f(x)$ 之间的一个一致偏离作为二维点的 $y$ 值。应该确信，点 $(x,y)$ 在比较函数 $f(x)$ 的面积下为均匀分布。

一个等价的过程就是，在 0 和 1 之间找出第二个一致偏离值，并根据它是否小于或大于比率 $p(x)/f(x)$ 来决定接受或拒绝它。

综上所述，对某个给定的 $p(x)$，拒绝方法要求人们断然地找到某个合理的比较函数 $f(x)$。此后每生成一个偏离值，要求有两个一致随机偏离值．一个是 $f$ 值的计算（以得到坐标 $y$），一个是 $p$ 值的计算（以决定是否接受或拒绝点 $x,y$）。图7.3.1表明了这一过程。当然这个过程平均说来重复 $A$ 次，然后才获得最终的偏离。




其中 p(x) 处处小于某个其它的函数 f(x)。变换方法首先用于生成分布 f(x) 的随机偏离 x（比较图7.2.1），第二个一致偏离用来决定是否接受或拒绝该 x。如果它被拒绝，就找 f 的一个新的偏离，以此类推。接受点与拒绝点的比率是，p 下的面积与 p 和 f 之间面积的比。

图7.3.1　由已知概率分布 $p(x)$ 生成随机偏离 $x$ 的拒绝方法

### 7.3.1　Γ(Gamma)分布

整数阶($a>0$)的 Γ 分布是，具有单位均值的泊松随机过程中第 $a$ 个事件的等待时间。例如，当 $a=1$时，它就是第7.2节的指数分布，即第一个事件的等待时间。

一个 Γ 偏离具有在 $x$ 和 $x+dx$ 之间出现 $x$ 值的概率 $p_a(x)dx$，其中

$$p_a(x)dx = \frac{x^{a-1}e^{-x}}{\Gamma(a)}dx \qquad x > 0 \tag{7.3.1}$$

为了对小的 $a$ 值生成式(7.3.1)的偏离，最好加上 $a$ 个指数分布等待时间，即一致偏离的对数。因为对数之和就是乘积的对数，人们实际上只须生成 $a$ 个一致偏离的乘积，然后取对数。

对大的 $a$ 值，分布(7.3.1)具有典型的"钟型"形式，在 $x=a$ 处有峰值，并且半宽度大约为 $\sqrt{a}$。

我们对具有同样性质形式的几个概率分布很感兴趣。对于这种情况的一个有用的比较函数由 **Lorentz 分布**推出。

$$p(y)dy = \frac{1}{\pi}\left(\frac{1}{1+y^2}\right)dy \tag{7.3.2}$$

它的不定积分的反函数恰好是正切函数。由此可得，对任何约束 $a_0$、$c_0$和 $x_0$，在比较函数

$$f(x) = \frac{c_0}{1+(x-x_0)^2/a_0^2} \tag{7.3.3}$$

面积下，一致随机点 $x$ 的坐标可由以下公式确定：

$$x = a_0\tan(\pi U) + x_0 \tag{7.3.4}$$

其中 $U$ 是 0 和 1 之间的一致偏离。因此，对某个特殊"钟型" $p(x)$概率分布，我们只需找出常数 $a_0$，$c_0$和 $x_0$，其中乘积 $a_0c_0$（它决定面积）应尽可能小，使得式（7.3.3）在任意点处处大于 $p(x)$。

Ahrens 已对 Γ 分布解决了这一问题，得出了如下算法（如 Knuth 书中描述[1]）：

```
#include <math.h>

float gamdev(int ia, long *idum)
使用 ran1(idum)作为一致偏离的来源，返回一个整数阶 ia 的 Γ 分布的偏离，即是在具有单位均值的泊松分布过程
中，对第 ia 个事件的等待时间。
{
    float ran1(long *idum);
    void nrerror(char error_text[]);
    int j;
    float am,e,s,v1,v2,x,y;

    if (ia < 1) nrerror("Error in routine gamdev");
    if (ia < 6) {                                   使用直接方法，加上等待时间
        x=1.0;
        for (j=1;j<=ia;j++) x *= ran1(idum);
        x = -log(x);
    } else {                                        使用拒绝方法
        do {
            do {
                do {                                这四行生成随机角的正切，即等价于
                    v1=2.0*ran1(idum)-1.0;              y = tan(π * ran1(idum))
                    v2=2.0*ran1(idum)-1.0;
                } while (v1*v1+v2*v2 > 1.0);
                y=v2/v1;
                am=ia-1;
                s=sqrt(2.0*am+1.0);
                x=s*y+am;                           我们决定是否拒绝 x：
            } while (x <= 0.0);                     在零概率区域内拒绝
            e=(1.0+y*y)*exp(am*log(x/am)-s*y);      概率fn与比较fn的比率
        } while (ran1(idum) > e);                   在第二个一致偏离基础上的拒绝
    }
    return x;
}
```

## 7.3.2 泊松偏离

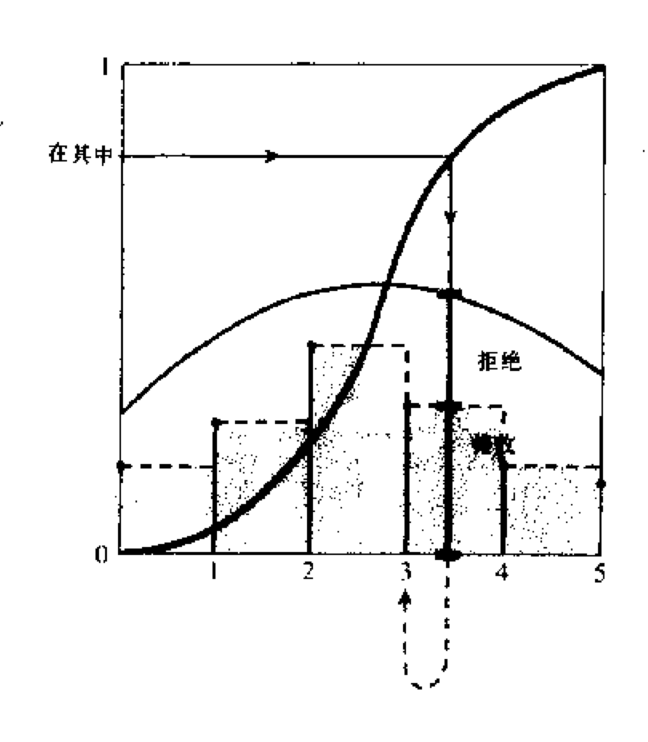
从概念上讲，泊松（Poisson）分布与 Γ 分布有关。泊松分布给出了在时间 $x$ 的给定区间内，某整数 $m$ 个单位速率的泊松随机事件出现的概率，而 Γ 分布是在 $x$ 和 $x+dx$ 之间对第 $m$ 个事件等待时间的概率。注意 $m$ 只取整数值≥0，所以将泊松分布看作一个连续分布函数 $p_x(m)dm$，除了 $m$ 取≥0的整数之外，泊松分布处处为零。在这样的区域里，它是无穷的，而使得在包含整数区域上的积分概率是某些有限数。在整数 $j$ 上的总概率为：

$$\text{Prob}(j) = \int_{j-\epsilon}^{j+\epsilon} p_x(m)dm = \frac{x^j e^{-x}}{j!} \tag{7.3.5}$$

初看上去，它未必是拒绝方法的候选分布，因为没有任何连续比较函数能够大于这个无穷高且又无限窄的 $p_x(m)$ 中的 **Dirac δ 函数**。不过我们有这样一个技巧：将 $j$ 处尖峰信号的有限面积均匀地扩展到 $j$ 和 $j+1$ 之间的区间中。这就定义了一个连续分布 $q_x(m)dm$，表示为：

$$q_x(m)dm = \frac{x^{[m]}e^{-x}}{[m]!}dm \tag{7.3.6}$$

其中$\lfloor m \rfloor$表示小于 $m$ 的最大整数。如果我们现在使用拒绝方法由式(7.3.6)生成一个(非整数)偏离，然后取该偏离的整数部分，它好象是由所求的分布式(7.3.5)得到的(参见图7.3.2)。这一技巧对任何整数概率分布是通用的。




方法是在表示成虚直线的阶梯函数上执行，产生一个实值偏离，这个偏离退到下一个较低的整数，这就是输出。

图7.3.2 作为应用到整数值分布的拒绝方法

对于足够大的 $x$，分布式(7.3.6)可定性地看成是“钟型”的(尽管是一个由小方步长制成的钟)，我们就可以采用前面已使过的同样类型的 Lorentz 变换。对小的 $x$，我们可以生成独立的指数偏离(事件之间的等待时间)；若这些偏离的和首次超出 $x$，那么，在等待时间 $x$ 内可能发生事件的个数就成为已知，并且它等于此和式的项数减一。

这些概念产生如下程序：

```
#include <math.h>
#define PI 3.141592654

float poidev(float xm, long *idum)
用 ran1(idum)作为一致随机偏离的来源，按浮点数返回整数值，它是从具有均值 xm 的泊松分布导出的随机偏离。
{
    float gammln(float xx);
```

```
float ran1(long *idum);
static float sq,alxm,g,oldm=(-1.0);          oldm 是自上一次调用后,xm 是否有所改变的标志
float em,t,y;

if (xm < 12.0) {                             使用直接方法
    if (xm != oldm){
        oldm=xm;
        g=exp(-xm);                          如果 xm 是新的,计算指数
    }
    em = -1;
    t=1.0;
    do{
        em += 1.0;                           替换叠加指数偏离,它等价于乘上一致偏离.
        t *= ran1(idum);                     实际上并不需要取对数,只不过比较一下事先计算出的指数
    } while (t > g);
} else{                                      使用拒绝方法
    if (xm != oldm){                         如果自下次调用后,xm 有所改变,
        oldm=xm;                             则预先计算下面出现的函数
        sq=sqrt(2.0*xm);
        alxm=log(xm);
        g=xm*alxm-gammln(xm+1.0);            函数 gammln 是如第6.1节中所给出的
    }                                        Γ函数的自然对数
    do{
        do{
            y=tan(PI*ran1(idum));            y 是来自 Lorentzian 比较函数的偏离
            em=sq*y+xm;                      em 是进行了位移并改变了尺度的 y
        } while (em < 0.0);                  若在零概率区域中则拒绝
        em=floor(em);                        整值分布的技巧
        t=0.9*(1.0+y*y)*exp(em*alxm-gammln(em+1.0)-g);
            所求的概率与比较函数的比,比较它和另一个一致偏离来决定是接受还是拒绝。因了0.9的选
    }while (ran1(idum) > t);                 择是使 t 不超出!
}
return em;
}
```

### 7.3.3 二项偏离

如果一个事件出现的概率为 $q$,并且我们作 $n$ 次试验,那么,它出现 $m$ 次的个数具有二项分布:

$$\int_{j-\epsilon}^{j+\epsilon} p_{n,q}(m)dm = \binom{n}{j} q^j(1-q)^{n-j} \tag{7.3.7}$$

二项分布是整数值的,在0到 $n$ 的可能值中选取 $m$ 次。它取决于两个参数 $n$ 和 $q$,所以它的实现相对于前面的情况更难一些。不过,已阐述的内容对完成这样的工作是不成问题的:

```
#include <math.h>
#define PI 3.141592654

float bnldev(float pp, int n, long *idum)
    用 ran1(idum)作为一致偏离的来源,按浮点数返回一整数值,该值是一个由 n 个试验(每个试验的概率为 pp)的二项
    式分布引出的随机偏离。
{
    float gammln(float xx);
    float ran1(long *idum);
    int j;
```

```
    static int nold=(-1);
    float am,em,g,angle,p,bnl,sq,t,y;
    static float pold=(-1.0),pc,plog,pclog,en,oldg;

    p=(pp<=0.5?pp:1.0-pp);              如果将答案也变为n减去它本身，则二项分布在pp到1－pp的替换之
                                         下是不变的，记住下面将这样做
    am=n*p;                              这是需要产生偏置的均值
    if(n<25){
        bnl=0.0;                         当n不是太大时，使用直接方法，这可能需要对ran1至多
        for(j=1;j<=n;j++)                25次调用
            if (ran1(idum) < p) bnl += 1.0;
    } else if(am < 1.0) {                如果在25次或更多次试验中，期望少于一次的事件发生
        g=exp(-am);                      则这个分布就是相当准确的泊松分布，使用直接泊松方法
        t=1.0;
        for (j=0;j<=n;j++) {
            t *= ran1(idum);
            if (t< g) break;
        }
        bnl=(j <= n ? j : n);
    else{                                使用拒绝方法
        if(n != nold){                   如果n改变，则计算有用的量
            en=n;
            oldg=gammln(en+1.0);
            nold=n;
        }if (p != pold) {                如果p改变，则计算有用的量
            pc=1.0-p;
            plog=log(p);
            pclog=log(pc);
            pold=p;
        }
        sq=sqrt(2.0*am*pc);              下面的代码现在看来应该很熟悉，这是具有Lorentz比较
        do {                             函数的拒绝方法
            do{
                angle=PI*ran1(idum);
                y=tan(angle);
                em=sq*y+am;
            }while (em < 0.0 || em >= (en+1.0));      拒绝
            em=floor(em);                             对于整数值分布的技巧
            t=1.2*sq*(1.0+y*y)*exp(oldg-gammln(em+1.0)
                -gammln(en-em+1.0)+em*plog-(en-em)*pclog);
        } while (ran1(idum) > t);                     拒绝，对每个偏离平均要发生大约1.5次
        bnl=em;
    }
    if (p!=pp) bnl=n-bnl;                             记住要取消对称变换
    return bnl;
}
```

对许多其他算法参阅Devroye[2]和Bratley[3]。

## 参考文献和进一步的读物：


Knuth, D. E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), pp. 120ff. [1]

Devroye, L. 1986, *Non-Uniform Random Variate Generation* (New York: Springer-Verlag), § x.4. [2]

Bratley, P., Fox, B. L., and Schrage, E. L. 1983, *A Guide to Simulation* (New York: Springer-Verlag). [3]

## 7.4 随机位的生成

C 语言可以提供一个很有用的方式，以进行机器水平的逐位运算，诸如<<(左移)。本节将阐明如何充分利用这种性能。

问题是，当 0 和 1 是等可能的情形，如何生成单个随机位。当然，可以只在零和一之间生成一致随机偏离，并使用它们的高阶位(也就是检验它们是否大于或小于0.5)。但这要进行很多运算。然而对某些特殊的应用，如实时信号处理，我们希望非常快地生成随机位

生成随机位的两个不同的实施方法是，基于“模2本原多项式”理论。这个理论的讨论超出本书范围。虽然第7.7节和第20.3节将讨论它的一些内容，而在这里，给出以下说明就足够了。存在这样一种特殊多项式，它们的系数不是 0 就是 1 。例如，

$$x^{18} + x^5 + x^2 + x^1 + x^0 \tag{7.4.1}$$

我们可简化而仅写出(系数)非零的 $x$ 的指数即：

$$(18,5,1,0)$$

每个 $n$ 阶的(上面是 $n=18$)模 2 本原多项式都定义了，由前 $n$ 个随机位得到一个新随机位的递推公式。这个递推公式保证可以产生一个极大长度的序列，也就是在它重复之前循环所有可能的 $n$ 位的序列(全部为 0 除外)。因而，人们可以得到具有任意初始模式的(全0除外)序列，并在序列重复之前得到 $2^n-1$ 个随机位。

将这些位从1(最近生成的)到 $n$($n$ 步以前生成的)进行编号，并记为 $a_1, a_2, \ldots, a_n$。我们要对一个新位 $a_0$给出公式。生成 $a_0$之后，我们将所有的位作移位一步，使旧的 $a_n$ 消失，并且新的 $a_0$成为 $a_1$。然后我们再次应用这公式，以此类推。

“方法 I ”最容易用硬件实施，它只需要一个 $n$ 位长的单个移位寄存器以及 $n$ 个 XOR 门(“异或”门或按位模2加)，C 语言的标志是“∧”，对以上给出的本原多项式，递推公式是：

$$a_0 = a_{18} \wedge a_5 \wedge a_2 \wedge a_1 \tag{7.4.2}$$

用∧连在一起的这些项可想象为移位寄存器上的“流出孔”，经异或后作为寄存器的输入。更一般地说，除去常数项(零位项)外，对本原多项式的每一个非零系数恰好都有一项与之对应。因而对一般 $n$ 阶本原多项式，第一项始终是 $a_n$，最后一项可能是 $a_1$或不是 $a_1$，它取决于本原多项式中是否有 $x^1$的项。

尽管用硬件实施很简单，但用 C 语言来实现方法 I 是相当繁琐的，因为所有位必须被一个全字屏蔽序列收集：

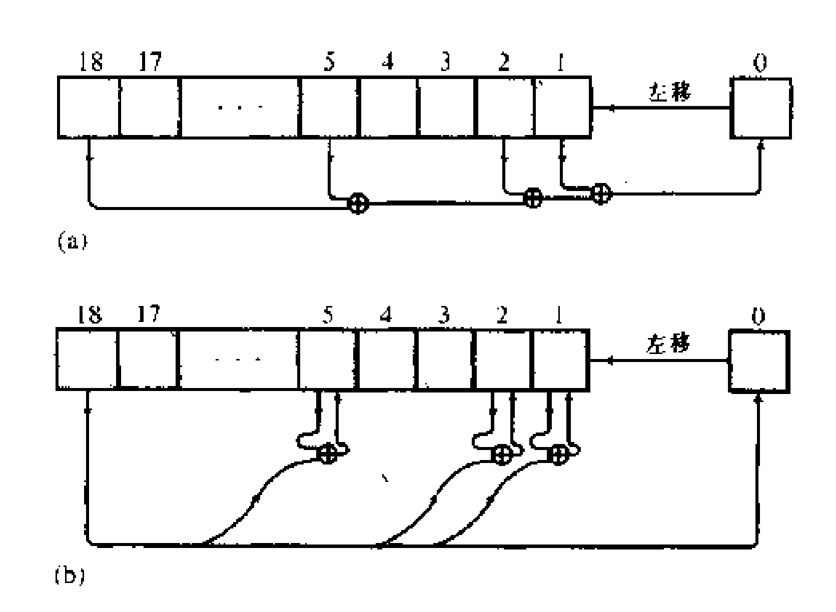
```
#define IB1 1;                          2的次幂
#define IB2 2;
#define IB5 16;
#define IB18 131072;

int irbit1(unsigned long *iseed)
    以整数返回一个随机位，它是以 iseed 中18个低有效位为基础的。(对下一次调用，iseed 被修改。)
{
    unsigned long newbit;                   累加的 XOR

    newbit = (*iseed & IB18) >> 17         得到位18
           ^ (*iseed & IB5) >> 4           具有位5的异或
           ^ (*iseed & IB2) >> 1           具有位2的异或
```

```
    ^ (*iseed & IB1));          只有位1的异或
  *iseed=(*iseed << 1) | newbit;  将种子值向左移位并在它的位1中放置异或的结果
  return (int) newbit;
}
```

“方法Ⅱ”不太适合直接以硬件实施(尽管仍有可能),却更适合C语言。随着每个新位生成,它在保存的 $n$ 个位中修正了不止一个的位(图7.4.1)。它能生成极大长度序列,但与“方法Ⅰ”顺序不同。




(a)所挑选的“流出孔”的内容由“异或”门(按位模2加)所组合,其结果从右边进行移位,这个方法最容易用硬件实施,(b)挑选出来的位通过它与最左边位的“异或”得以修正,然后最左边的位从右边移入,这个方法最容易用软件实施。

图7.4.1 从移位寄存器和模2本原多项式得到随机位的两个有关的方法

对于本原多项式(7.4.1)应有以下关系式:

$$a_0 = a_{18}$$
$$a_5 = a_5 \wedge a_0$$
$$a_2 = a_2 \wedge a_0 \qquad (7.4.3)$$
$$a_1 = a_1 \wedge a_0$$

一般在本原多项式,除0次项和 $n$ 次项外,对每个非零项都要作一次异或操作,方法Ⅱ的优点是,所有的异或操作通常都可由单个屏蔽字的异或操作来完成。

```
#define IB1 1;
#define IB2 2;
#define IB5 16;
#define IB18 131072;
#define MASK (IB1+IB2+IB5);

int irbit2(unsigned long *iseed)
  按整数返回一个随机位,它是以iseed中18个低有效位为基础的(对下一次调用,iseed被修改。)
{
  if (*iseed & IB18) {          改变所有屏蔽位,位移,并将1放进位1中
    *iseed=((*iseed ^ MASK) << 1) | IB1;
    return 1;
  } else {                      位移并将0放进位1中
```

```
        *iseed <<= 1;
        return 0;
    }
}
```

告诫：不要使用这些程序的连续位，用作为一个很大而且猜想为随机的整数位，或者，用它作为一个猜想为随机浮点数的尾数位。在该情况下，它们并不是十分随机的。参见Knuth[1]。这些随机位可利用的实例是：(i)以快速的"芯片速率"随机地用±1乘上一个信号，使得它的谱在一给定的带通内，一致地（可恢复）展开，或者(ii)二元树的蒙特卡罗探测，其中，究竟向左或向右分支的决定是随机作出的。

**表7.4.1　模2本原多项式**

| | |
|---|---|
| (1, 0) | (51, 6, 3, 1, 0) |
| (2, 1, 0) | (52, 3, 0) |
| (3, 1, 0) | (53, 6, 2, 1, 0) |
| (4, 1, 0) | (54, 6, 5, 4, 3, 2, 0) |
| (5, 2, 0) | (55, 6, 2, 1, 0) |
| (6, 1, 0) | (56, 7, 4, 2, 0) |
| (7, 1, 0) | (57, 5, 3, 2, 0) |
| (8, 4, 3, 2, 0) | (58, 6, 5, 1, 0) |
| (9, 4, 0) | (59, 6, 5, 4, 3, 1, 0) |
| (10, 3, 0) | (60, 1, 0) |
| (11, 2, 0) | (61, 5, 2, 1, 0) |
| (12, 6, 4, 1, 0) | (62, 6, 5, 3, 0) |
| (13, 4, 3, 1, 0) | (63, 1, 0) |
| (14, 5, 3, 1, 0) | (64, 4, 3, 1, 0) |
| (15, 1, 0) | (65, 4, 3, 1, 0) |
| (16, 5, 3, 2, 0) | (66, 8, 6, 5, 3, 2, 0) |
| (17, 3, 0) | (67, 5, 2, 1, 0) |
| (18, 5, 2, 1, 0) | (68, 7, 5, 1, 0) |
| (19, 5, 2, 1, 0) | (69, 6, 5, 2, 0) |
| (20, 3, 0) | (70, 5, 3, 1, 0) |
| (21, 2, 0) | (71, 5, 3, 1, 0) |
| (22, 1, 0) | (72, 6, 4, 3, 2, 1, 0) |
| (23, 5, 0) | (73, 4, 3, 2, 0) |
| (24, 4, 3, 1, 0) | (74, 7, 4, 3, 0) |
| (25, 3, 0) | (75, 6, 3, 1, 0) |
| (26, 6, 2, 1, 0) | (76, 5, 4, 2, 0) |
| (27, 5, 2, 1, 0) | (77, 6, 5, 2, 0) |
| (28, 3, 0) | (78, 7, 2, 1, 0) |
| (29, 2, 0) | (79, 4, 3, 2, 0) |
| (30, 6, 4, 1, 0) | (80, 7, 5, 3, 2, 1, 0) |
| (31, 3, 0) | (81, 4, 0) |
| (32, 7, 5, 3, 2, 1, 0) | (82, 8, 7, 6, 4, 1, 0) |
| (33, 6, 4, 1, 0) | (83, 7, 4, 2, 0) |
| (34, 7, 6, 5, 2, 1, 0) | (84, 8, 7, 5, 3, 1, 0) |
| (35, 2, 0) | (85, 8, 2, 1, 0) |
| (36, 6, 5, 4, 2, 1, 0) | (86, 6, 5, 2, 0) |
| (37, 5, 4, 3, 2, 1, 0) | (87, 7, 5, 1, 0) |
| (38, 6, 5, 1, 0) | (88, 8, 5, 4, 3, 1, 0) |
| (39, 4, 0) | (89, 6, 5, 3, 0) |
| (40, 5, 4, 3, 0) | (90, 5, 3, 2, 0) |
| (41, 3, 0) | (91, 7, 6, 5, 3, 2, 0) |
| (42, 5, 4, 3, 2, 1, 0) | (92, 6, 5, 2, 0) |
| (43, 6, 4, 3, 0) | (93, 2, 0) |
| (44, 6, 5, 2, 0) | (94, 6, 5, 1, 0) |
| (45, 4, [illegible] 1, 0) | (95, 6, 5, 4, 2, 1, 0) |
| (46, 8, 5, 3, 2, 1, 0) | (96, 7, 6, 4, 3, 2, 0) |
| (47, 5, 0) | (97, 6, 0) |
| (48, 7, 5, 4, 2, 1, 0) | (98, 7, 4, 3, 2, 1, 0) |
| (49, 6, 5, 4, 0) | (99, 7, 5, 4, 0) |
| (50, 4, 3, 2, 0) | (100, 8, 7, 2, 0) |

现在，不必仔细研究前面的例子中 18 次本原多项式具有哪些独特之处。(我们之所以选择 18，是为了直接用数值实验证实我们的陈述，而$2^{18}$已足够小了)。上页表7.4.1列出一些模 2 的本原多项式，它阶数高至 100。(事实上，每一阶都存在一些本原多项式，例如，可参阅第7.7节高至 10 阶的完整表。)

参考文献和进一步读物：


Knuth, D. E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), pp. 29ff. [1]

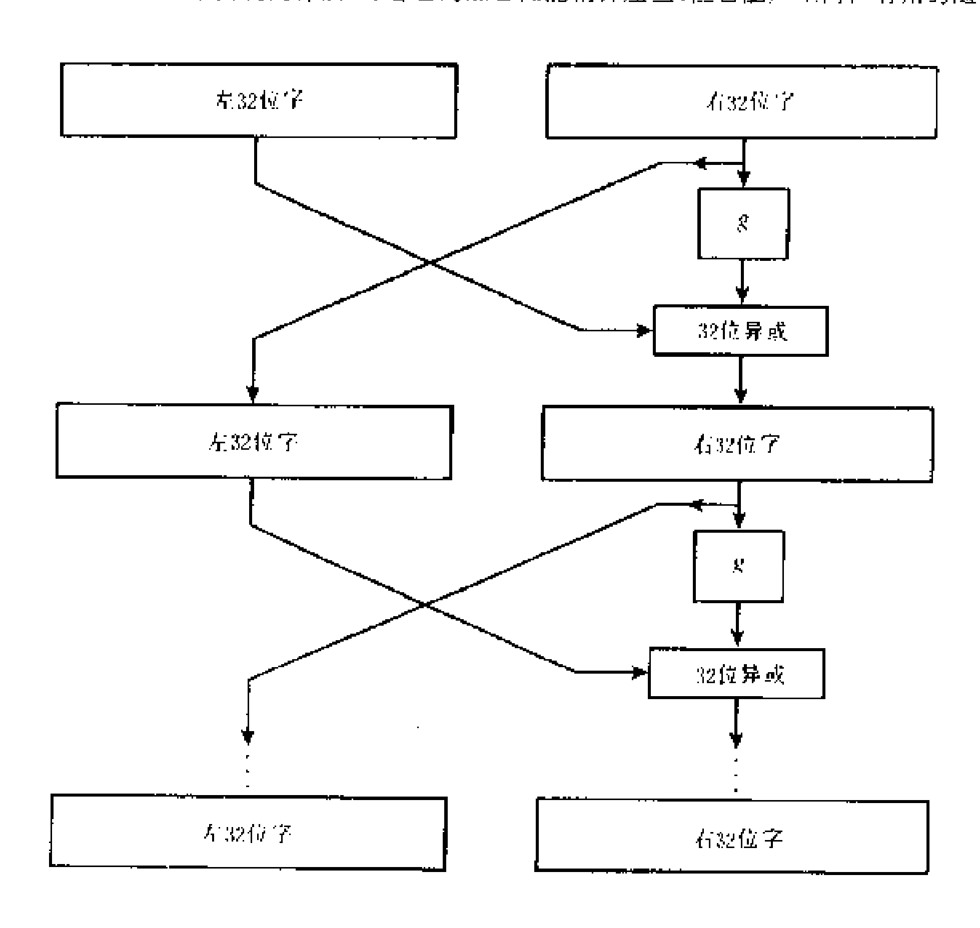
Horowitz, P., and Hill, W. 1989, *The Art of Electronics*, 2nd ed. (Cambridge: Cambridge University Press), §§9.32~9.37.

Tausworth, R.C. 1965, *Mathematics of Computation*, vol. 19, pp. 201~209.

Watson, E.J. 1962, *Mathematics of Computation*, vol. 16, pp. 368~369. [2]


## 7.5 基于数据加密的随机序列

在本书的第一版本里，我们讲述了如何利用数据加密标准(DES)[1-3]来产生随机数。遗憾的是，当用高级语言，如 C 语言来实现软件时，数据加密标准慢的难以让人接受。其实，上述这种实施明显地看出是弊大于利。在这里，我们给出一个简捷的算法，尽管它的加密性能稍许差些，但它能产生同样有用的随机数。




对两组32位码字进行非线性函数 $g$ 的迭代(*Meyer* 和 *Matyas*[4])。

图7.5.1 以图中方式显示了数据加密标准(DES)

数据加密标准、像它的先前的加密系统 LUCIFER 一样，就是通常所说的"分组乘积加密"[1]，以迭代方式(16 次)采用一种高度非线性的位混合操作，作用于 64 位输入。上页图7.5.1表示出这种混合过程的 DES 信息流程。函数 $g$ 称为"密码函数"，它接受 32 位，输出 32 位。Meyer 和 Matyas[2]论述了这种密码函数的重要性在于它的非线性以及其他设计标准。

数据加密标准，用一个复杂的作用于短序列的连续位的查找表，和位的置换系统来构成它的密码函数 $g$。显然，这个函数有很强的加密功能(或许如某些批评家所争论的那样，有极微妙难辨认的保密缺陷!)。为了我们的目的，最好采用另一种函数 $g$，它在用高级语言时能快速计算，这种函数可能会减弱一些算法的加密性，但我们的目的不仅是加密；我们还要最快地找到函数 $g$，迭代最少的混合过程(图7.5.1)，这个可通过标准测试的输出随机序列，经常可用来产生随机数。这个合成的算法将不是数据加密标准，而多半是一种"伪随机加密标准"，它比较适合我们目的。

按上述准则，函数 $g$ 应是非线性的。我们在函数 $g$ 中把整数乘法操作放在一个重要的位置，因为，在高级语言中，一般是不采用 64 位寄存器，必须把 16 位乘法操作限制在 32 位的结果中。因而，关于函数 $g$ 的大致构思是，必须分别计算输入半字的高 16 位、低 16 位的三个乘法，得出三个不同 32 位乘积，然后，经过快速运算(如用"加"或者"异或"运算)组合它们，或许再加上个固定常数，成为一个 32 位结果。

这里几个一般方案均有效实现方法，可选择其中之一进行系统研究。对输出随机性的测试和实验得出图7.5.2所示操作步骤。图中几个新元件需要解释一下：值 $C_1$ 与 $C_2$ 是固定常数，它们是随机选择的，但其中必须准确包含有 16 位和 16 位 0 。通过异或操作组合这些常数以保证整个 $g$ 没有倾向性偏 0 位或 1 位。

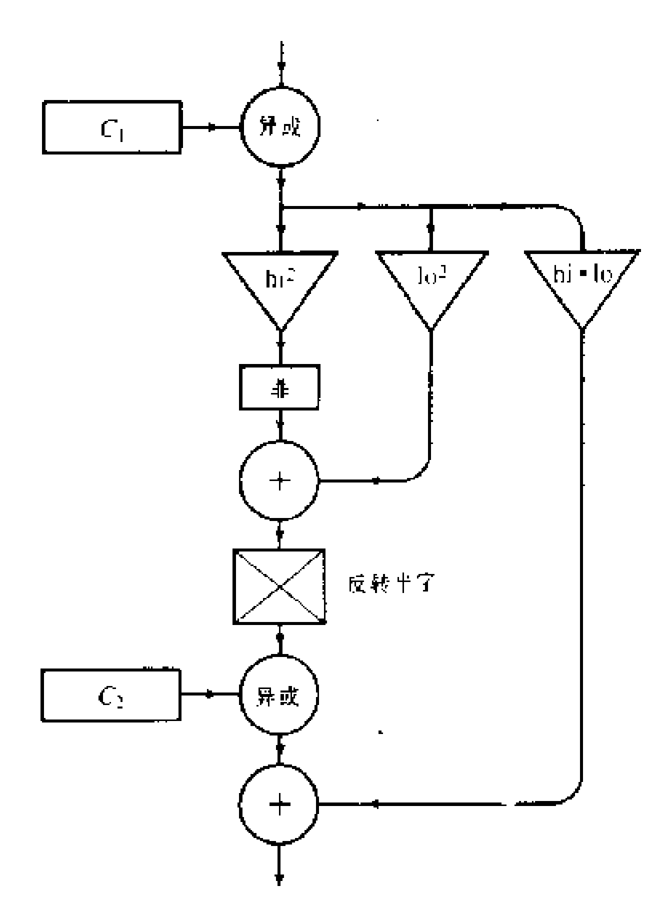



图7.5.2 程序 **Psdes** 使用非线性函数 $g$。

在图7.5.2中，弄清楚"反转半字"操作是非常重要的，否则最低位与最高位将不会被三个乘积适当地

混洗。因而在 $g$ 中隐含的几项选择是：在纵向"流水线"的什么地方做反转？按什么次序组合这三个乘积与常数 $C_2$？每一个组合都需要进行"加"及"异或"操作吗？在确定图中算法之前，对这些选择需要多做一些试验。

最少迭代次数 $N_{it}$ 的确定可以简略，有意义的 $N_{it}$ 最小值显然是2，因为一次迭代只能简单地移动一个32位而不会改变它。可以利用常数 $c_1$ 和 $c_2$ 来帮助确定合适的 $N_{it}$ 值；当 $N_{it}=2$，且 $C_1=C_2=0$（有意粗劣的选择），生成程序使一些随机性的测试失败，另一方面，当 $N_{it}=4$ 或 $N_{it}=2$，但常数 $C_1$ 和 $C_2$ 不太大，这时从这个结构得到的相当于$10^9$个浮点数 $r_i$ 的序列里，我们还没有发现对随机性的任何统计偏差。因而 $N_{it}=4$ 及不太小的 $C_1$，$C_2$ 的组合给出的序列是随机的，对其随机性的测试已远超出我们的实际实验。在 $N_{it}=2$（当然快两倍）时，我们并没有发现它的不随机性，尽管事实如此，我们还是推荐使用较为保守的参数值（$N_{it}=4$）。

这些思想的实现简单易懂。下述程序不是可随便移植的，因为，已设无符号长整数是 32 位，是大多数机器的实际情形。然而，并不是说较长的整数总是不好的（其常数 $C_1$，$C_2$ 有适当的范围）。C 语言并不提供便于移植的方法，以便分割一个长整数为半字，故我们必须使用屏蔽的组合（& 0xffff），并左移或右移 16 位（<< 16 或>>16）。在某些机器上半字提取比使用 C 逻辑与结构要快得多，但这在"长结尾"与"短结尾"机器之间一般是不可移植的（长、短结尾指的是，将字节存放到一个字里的次序）。

```
#define NITER 4

void psdes(unsigned long *lword,unsigned long *irword)
    混洗的64位字长(lword,irword)的"伪随机数据加密标准"。这二个32位变量在所有位都被反转混乱了.
{
    unsigned long i,ia,ib,iswap,itmph=0,itmpl=0;
    static unsigned long c1[NITER]={
        0xbaa96887L,0x1e17d32cL,0x03bcdc3cL,0x0f33d1b2L};
    static unsigned long c2[NITER]={
        0x4b0f3b58L,0xe874f0c3L,0x6955c5a6L,0x55a7ca46L};

    for (i=0;i<NITER;i++) {                    实现 DES 逻辑运算的 niter 次迭代，采用简单的，非加密的
                                               非线性函数替换 DES
        ia=(iswap=(*irword)) ^ c1[i];          常数 c1 和 c2(下面)保证了非线性混合
        itmpl = ia & 0xffff;
        itmph=ia>>16;
        ib=itmpl*itmpl+ ~(itmph*itmph);
        *irword=(*lword) ^ (((ia=(ib >> 16)|
            ((ib & 0xffff) << 16)) ^ c2[i])+itmpl*itmph);
        *lword=iswap;
    }
}
```

列在下面的程序 ran4使用 psdes 以产生均匀的随机偏差。我们采用习惯做法，将变量 idum 的负值放在左32位，而正值 $i$ 放在右32位，返回第 $i$ 个随机偏差，及 idum 到 $i+1$ 的增量。这只是为了便于定义许多不同的序列(idum 的负值)还要随机存取每一个序列(idum 的正值)。为了从32位整数中得到一个浮点数，我们采用了第7.1.4节末尾叙述的掩蔽技巧，十六进制常数3F800000和007FFFFF 适合于用 IEEE 表示的计算机的32位浮点(如 IBM-PC 机和大多数 UNIX 工作站)，对 DEC-VAX 机，正确的十六进制常数分别为00004084和 FFFFC07F。若要扩大可移植性，可以用有符号非负的32位整数来构成一个浮点数(典型地，如是负数，则刚好加$2^{31}$)，然后用一个浮点常数乘它(典型地，为$2^{-31}$)。

程序 **ran4**一个有趣的，有时也很有用处的特点是，它允许在序列中随机访问第 $n$ 个随机数值，而不需要先生成 $1\cdots n-1$ 个数值。这个性质是由于共享了任何基于混洗的随机数生成程序(数值高度群集的数据键的映射技术，几乎均匀地进入一个存储地址空间)[5,6]。有时可能会碰到一个少见的伪装掩盖的问题，当在某些个别地方发生后，最重要的是结果应可以辨认出来。我们希望使用相同的随机值，也就是说改变某些控制参数，使这种情况能模拟返回并重新开始。这类的问题诸如"在循环数 337098901 中使用了什么随

机数?",然而,在提出这个问题之前它可能已经是循环数为395100273了。基于递归的随机数生成程序比基于混洗的随机数生成程序更不易回答这类问题。

```
float ran4 (long *idum)
```

返回在0.0到1.0范围内一致随机偏离。它由64位字(idums,idum)经伪-DES(与DES类似)混洗而生成的,其中idums是用负值idum前次调用的设置,并增加idum,通过连续调用,本程序可用来产生随机序列,而在连续调用之间保持idum不变;当用idum=n来调用时,本程序能够在一序列中随机访问第n位偏离值。当用不同的idum的负值调用,则不同的序列被初始化。

```
{
    void psdes(unsigned long *lword, unsigned long *irword);
    unsigned long irword,itemp,lword;
    static long idums=0;
#if defined(vax) || defined(_vax_) || defined(__vax__) || defined(VAX)
    static unsigned long jflone=0x00004080;     十六进制常数jflone和jflmsk是用于1.和2.之间产生浮
    static unsigned long jflmsk=0xffff007f;     点数。它们与机器有依赖关系。参阅正文
#else
    static unsigned long jflone=0x3f800000;
    static unsigned long jflmsk=0x007fffff;
#endif

    if (*idum<0) {                              重新设置idums并且准备返回序列中的第一个偏离值
        idums = -(*idum);
        *idum=1;
    }
    irword=(*idum);
    lword=idums;
    psdes(&lword,&irword);                      用"伪-DES"对字编码
    itemp=jflone | (jflmsk & irword);           将浮点数掩蔽在1至2之间
    ++(*idum);
    return (*(float *)&itemp)-1.0;              相减后移到0.至1的范围
}
```

下表7.5.1给出一些数据,以检验用户的机器上**ran4**和**psdes**是否能正常工作。我们并不推荐应用**ran4**,除非你能复制所示的十六进制的数值。一般地,**ran4**比**ran0**(第7.1节)大约慢4倍,比**ran1**慢3倍。

**表7.5.1　检验psdes能否实行的数据**

| 检验psdes能否实行的数据 | | | | | | |
|---|---|---|---|---|---|---|
| idum | psdes调用前 | | psdes调用后(十六进制) | | ran4(idum) | |
| | lword | irword | lword | irword | VAX | PC |
| −1 | 1 | 1 | 604D1DCE | 509C0C23 | 0.275898 | 0.219120 |
| 99 | 1 | 99 | D97F8571 | A66CB41A | 0.208204 | 0.849246 |
| −99 | 99 | 1 | 7822309D | 64300984 | 0.034307 | 0.375290 |
| 99 | 99 | 99 | D7F376F0 | 59BA89EB | 0.838676 | 0.457334 |

逐次用变量−1,99,−99,1调用psdes,应得到表中显示的正确的lword与irword数值。掩模转换返回的浮点随机值将由机器决定;对于VAX和PC机表中示显示了数值

## 参考文献和进一步读物:


*Data Encryption Standard*, 1977 January 15, Federal Information Processing Standards Publication, number 46 (Washington: U.S. Department of Commerce, National Bureau of Standards). [1]

*Guidelines for Implementing and Using the NBS Data Encryption Standard*, 1981 April 1, Federal Information Processing Standards Publication, number 74 (Washington: U.S. Department of Commerce, National Bureau of Standards). [2]

*Validating the Correctness of Hardware Implementations of the NBS Data Encryption Standard*, 1980, NBS Special Publication 500—20(Washington:U. S. Department of Commerce, National Bureau of Standards). [3]

Meyer, C. H. and Matyas, S. M. 1982, Cryptography:*A New Dimension in Computer Data Security*(New York:Wiley). [4]

Knuth, D. E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming*(Reading, MA:Addison-Wesley), Chapter 6. [5]

Vitter, J. S. and Chen, W-C 1987, *Design and Analysis of Coalesced Hashing* (New Youk:Oxford University Press). [6]

## 7.6 简单的蒙特卡罗积分

从不稳定的信源能产生出数值方法这件事中受到启发。如,"样条"是由绘图员首次用于木料的任意条纹。"模拟退火"(将在第10. 9节讨论)是来源于热力学模拟。更何况大家都知道,至少耳闻过"蒙特卡洛积分方法"这个名字。

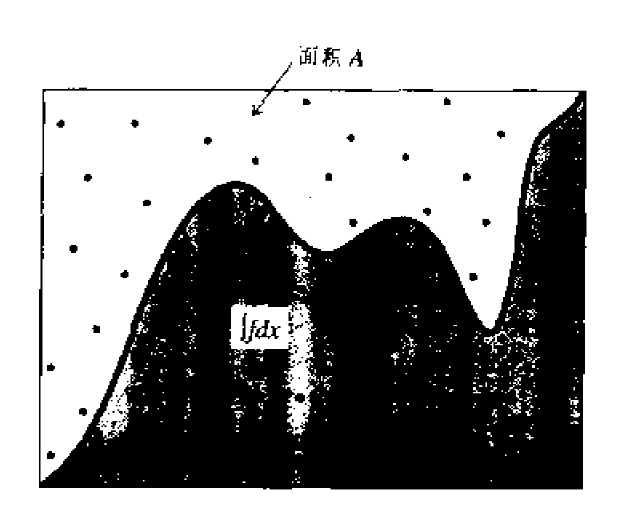
假设我们在一个多维体积 $V$ 中,选取 $N$ 个均匀分布的随机点,称它们为 $x_1,\cdots,x_N$。于是,蒙特卡罗(Monte Carlo)积分的基本定理指出,函数 $f$ 在多维体积上积分近似为:

$$\int f\,dV \approx V\langle f\rangle + V\sqrt{\frac{\langle f^2\rangle - \langle f\rangle^2}{N}} \tag{7.6.1}$$

这里,角括号〈 〉表示对 $N$ 个取样点上的算术平均值,

$$\langle f\rangle = \frac{1}{N}\sum_{i=1}^{N} f(x_i) \qquad \langle f^2\rangle \equiv \frac{1}{N}\sum_{i=1}^{N} f^2(x_i) \tag{7.6.2}$$

式(7. 6. 1)中的"加或减"项表示积分的标准偏差,而不是严格的界。进一步说,它不保证误差服从高斯分布,所以此误差项只可看成为可能误差的粗略估计。




随机点在区域 $A$ 中选取,函数 $f$ 的积分近似为曲线 $f$ 下随机点的部分乘上 $A$ 的面积,这个过程的重新细分可以改进这个方法的准确性,见正文。

图7. 6. 1 蒙特卡罗积分

假设要对函数 $g$ 在一个不容易随机取样的区域 $W$ 上积分。例如,$W$ 可能具有非常复杂

的形状。毫无疑问，只需找一个既包含 $W$ 又能够很容易取样的区域 $V$（上页图7.6.1），然后定义函数 $f$，它在 $W$ 内等于 $g$，在 $W$ 外（但仍在取样的 $V$ 中）等于零。取 $V$ 尽可能靠近 $W$，因 $f$ 的零值会增加式(7.6.1)的误差估计项。最好是，$W$ 之外所选择的取样点没有信息内容，使有效点数 $N$ 减少。式(7.6.1)中的误差估计项要考虑这个问题。

给出蒙特卡罗积分的一个通用程序是相当复杂(见第7.8节)，而一个实例将表明这种方法的简要原理。假设我们要对一个形状复杂的物体，例如一个由大方框定界的圆环的交，寻找它的重量和质心的位置。具体说，设该物体受如下三个联合条件限制：

$$z^2 + \left(\sqrt{x^2 + y^2} - 3\right)^2 \leq 1 \tag{7.6.3}$$

（圆环中心位于原点，长半径=4，短半径=2。）

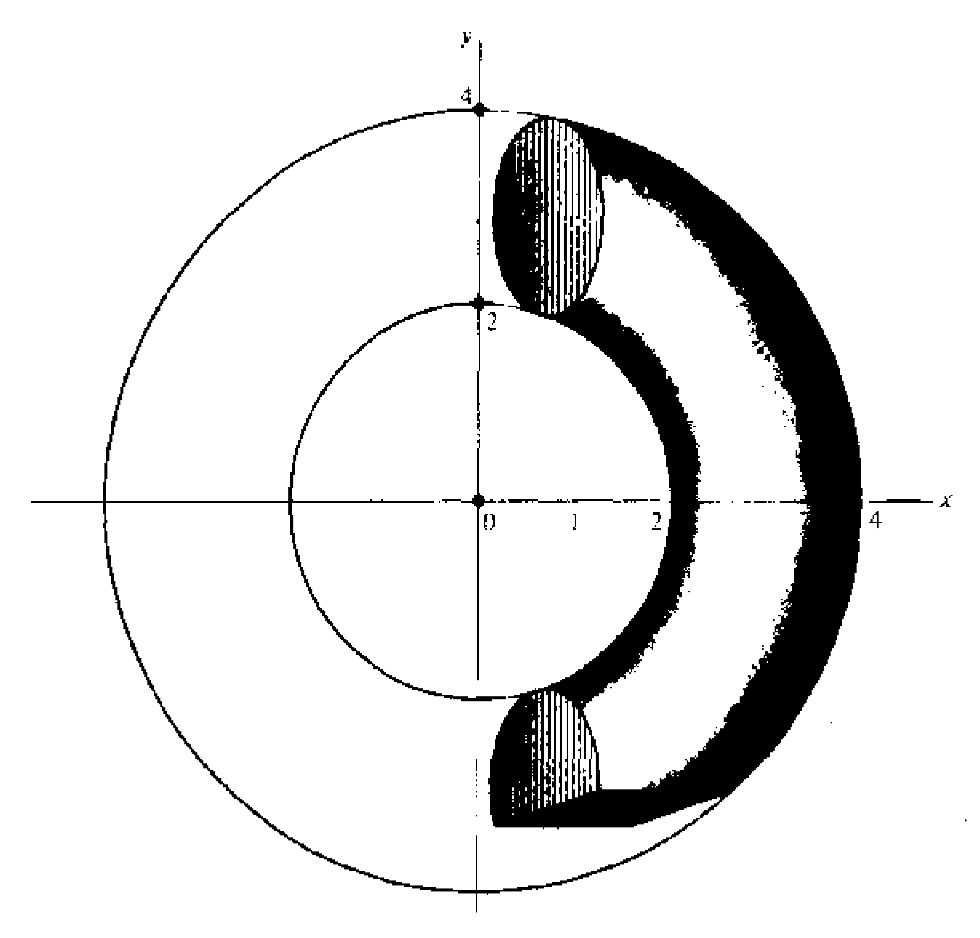
$$x > 1 \qquad y > -3 \tag{7.6.4}$$

（方框的两个面，见图7.6.2）。假设对矩量而言，物体具有定常密度 $\rho$。

我们要估计如下复杂物体内部的积分：

$$\int \rho\, dx\, dy\, dz \qquad \int x\rho\, dx\, dy\, dz \qquad \int y\rho\, dx\, dy\, dz \qquad \int z\rho\, dx\, dy\, dz \tag{7.6.5}$$

其质心坐标是后三个积分(线性矩量)与第一个积分(重量)之比。




积分区域是一段圆环，由两个平面的交界限定。区域的界限不容易写成解析封闭形式，所以蒙特卡罗是一个有用的方法。

图7.6.2 蒙特卡罗积分的例子(见正文)

在下面的程序段中，包围圆环段 $W$ 的区域 $V$ 是一个长方箱体，其范围是 $x=1\sim4$，$y=-3\sim4$，$z=-1\sim1$。

```
#include "nrutil.h"
...
n=...                                   设置所求的样本点个数
den=...                                 设置密度的常数值
sw=swx=swy=swz=0.0;                     零化累加的各种和式
varw=varx=vary=varz=0.0;
vol=3.0*7.0*2.0;                        取样区域的体积
for(j=1;j<=n;j++) {
    x=1.0+3.0*ran2(&idum);              在取样区域中随机挑选一点
    y=(-3.0)+7.0*ran2(&idum);
    z=(-1.0)+2.0*ran2(&idum);
    if (z*z+SQR(sqrt(x*x+y*y)-3.0) < 1.0) {      它在圆环中吗?
        sw += den;                      如果在圆环中，则加上各种累积量
        swx += x*den;
        swy += y*den;
        swz += z*den;
        varw += SQR(den);
        varx += SQR(x*den);
        vary += SQR(y*den);
        varz += SQR(z*den);
    }
}
w=vol*sw/n;                             积分式(7.6.5)的值
x=vol*swx/n;
y=vol*swy/n;
z=vol*swz/n;
dw=vol*sqrt((varw/n-SQR(sw/n))/n);      及其对应的误差估计
dx=vol*sqrt((varx/n-SQR(swx/n))/n);
dy=vol*sqrt((vary/n-SQR(swy/n))/n);
dz=vol*sqrt((varz/n-SQR(swz/n))/n);
```

在蒙特卡罗积分中，变量的替换是很有用的，例如，假设要计算同样的积分，但只是圆环段的密度依赖于坐标 $z$ 而变化，事实上，它按照公式

$$\rho(x,y,z)=e^{5z} \tag{7.6.6}$$

变化。计算这个积分的一个方法是，刚好是在第一次使用 den 之前，将语句

```
den=exp(5.0*z)
```

嵌在 if(……)块中。这样是可行的，但不是最佳方法。因为式(7.6.6)随 $z$ 下降(降到它的下限$-1$)迅速到零，这时大多数取样点对重量或矩量的和式几乎无贡献。这些点实际上浪费掉了，糟糕的就象这些点都位于区域 $W$ 之外似的。而变量替换，如同第7.2节的交换方法所做的，能解决这个问题。令：

$$ds=e^{5z}dz \quad 使得 \quad s=\frac{1}{5}e^{5z},\quad z=\frac{1}{5}\ln(5s) \tag{7.6.7}$$

则 $\rho\,dz=ds$，由界限 $-1<z<1$，变成界限 $1.00135<s<29.682$，现在程序段如下：

```
#include "nrutil.h"
...
n=...                                   设置需求的样本点个数
sw=swx=swy=swz=0.0;
varw=varx=vary=varz=0.0;
ss=0.2*(exp(5.0)-exp(-5.0))             需要随机取样的s区间
vol=3.0*7.0*ss                          在x,y,s空间中的体积
```

```
for(j=1;j<=n;j++) {
    x=1.0+3.0*ran2(&idum);
    y=(-3.0)+7.0*ran2(&idum);
    s=0.00135+ss*ran2(&idum);          挑选 s 中的一点
    z=0.2*log(5.0*s);                  方程(7.6.7)
    if (z*z+SQR(sqrt(x*x+y*y)-3.0) < 1.0) {
        sw += 1.0;                     因为考虑了 s 的定义，权重是 1
        swx += x;
        swy += y;
        swz += z;
        varw += 1.0;
        varx += x*x;
        vary += y*y;
        varz += z*z;
    }
}
w=vol*sw/n;                            积分(7.6.5)的值
x=vol*swx/n;
y=vol*swy/n;
z=vol*swz/n;
dw=vol*sqrt((varw/n-SQR(sw/n))/n);     及式对应的误差估计
dx=vol*sqrt((varx/n-SQR(swx/n))/n);
dy=vol*sqrt((vary/n-SQR(swy/n))/n);
dz=vol*sqrt((varz/n-SQR(swz/n))/n);
```

稍加思考就可看出，式(7.6.7)之所以有用，是因为我们那时想要消去($e^{5z}$)的被积函数部分是解析可积的，并且有一个能够解析求逆的积分(比较第7.2节)。一般情况，这些性质不具备。怎么办呢？答案是去掉被积函数可以求积分和求逆的“最佳”因子。“最佳”的准则就是，试图把剩下的被积函数约成一个尽可能靠近常数的函数。

极限情形是有启发性的：如果设法使得被积函数 $f$ 准确为常数，如果已知体积的区域 $V$ 准确包住所求的区域 $W$，则计算的 $f$ 的平均值将准确为它的常数值，并且式(7.6.1)中误差估计准确地为零。事实上，我们将能准确地求得积分，蒙特卡罗数值求值却成了多余的。所以，由极端情形稍作后退，以达到能用变量替换使函数 $f$ 近似为常数的程度，并且达到能够在一个只稍比 $W$ 大的区域内取样的程度，这就能提高蒙特卡罗的积分的准确性。这种技术在文献中一般叫做**方差退化**。

蒙特卡罗积分的基本缺点是，它的准确性只随样本点个数 $N$ 的平方根增长。如果对准确性要求适中，或者计算机的预算量很大，那么这个技术作为极为普遍的技术是很值得推荐的。

在后面的两节中，我们将了解到“消除 $N$ 的平方根这一障碍”的现有技术和成果，至少在某些场合，少数几个函数的计算具有较高的精确度。

参考文献和进一步读物：


Sobol, I. M. 1974, *The Monte Carlo Method* (Chicago: University of Chicago Press).

Kalos, M. H., and Whitlock, P. A. 1986, *Monte Carlo Methods* (New York: Wiley).


## 7.7 准随机序列

我们已经知道，在 $N$ 维空间中，均匀随机地选择 $N$ 个点会，在蒙特卡罗积分中引起一个误差项，它以 $1/\sqrt{N}$ 形式下降。大体上，每一个新的取样点都线性地加到累加的和上，并

成为函数平均值;同时也线性地加到累加的平方和上,并成为方差(方程7.6.2)。预计误差来自这个方差的平方根,即幂 $N^{-1/2}$。

这个平方根的收敛性是常见的,但并非总是如此。一个简单的反例是,在笛卡尔栅格上选择取样点,一个栅格取一点(无论什么次序)。于是,蒙特卡罗方法成为一个确定的求积结构——尽管是一个简单的——它的相对误差下降至少有 $N^{-1}$ 那样快(如果函数在取样区域的边界上平滑趋于零,或在此区域内是周期性函数,则它下降得还要更快)。

有关栅格的一个问题是,必须预先决定它应有什么精度,然后完成全部取样点。用栅格做取样,直到收敛或遇到临界时终止,这是不方便的。有人也许会问,是否存在有某种中间方案,随机地拾取样点的方法,并且扩展出某种自身回避的方法,它能回避均匀随机点的群集的机会发生。

除此以外,还有一种类似的问题。我们可能要为某个点而搜索一个 N 维空间,那里(局部可计算)的条件是满足的。当然,为使计算有意义,最好空间必须是连续的,以使在一个有限的 $N$ 维邻域内,所求的条件是满足的。尽管我们可能事先不知道这个邻域究竟有多大,但我们将平滑地更加细分尺度以增加取样点,直到发现所求的点为止。是否还存在比不相关的随机取样法更好的方法能完成这件事情呢?

对上面问题的回答是"是的"。比不相关的随机点更均匀地充满 $N$ 维空间的 $N$ 元序列,被称为**"准随机序列"**。这个术语有点用词不当,因为准随机序列根本不"随机";事实上,它们很明显地是次随机。严格说,在准随机序列中,取样点是尽可能的互相"最大地回避"。

一个简单的、概念上的例子是**霍尔顿(Halton)序列**[1]。在一维空间,这个序列第 $j$ 个数 $H_j$ 按以下步骤得到:(1)写出 $j$ 作为以 $b$ 为基的数,$b$ 为某个素数。(例如,以 $b=3$ 为基数,$j=17$ 时得出以 3 为基的数为 122)。(2)在这个序列前,反转这个数,并置一个小数点(即以基数 $b$ 的十进制小数点。例如,得基 3 的 0.221),所得结果就是 $H_j$。为在 N 维空间得到 $N$ 元序列,可以用不同的素数为基数 $b$,做成霍尔顿序列的每一个成份。一般情况采用前 $N$ 个素数。

不难看出霍尔顿序列是怎样工作的:每次数字 $j$ 的个数增加一位,$j$ 的反转数字的尾数变成了精细网格 b 的一个因子。因此这个过程是在越来越精细的笛卡尔栅格中填充所有点的每一个点——并且在每一栅格上最大地扩展数量级(因为,数字 $j$ 中最急剧变化的数,控制着最有意义的小数)。

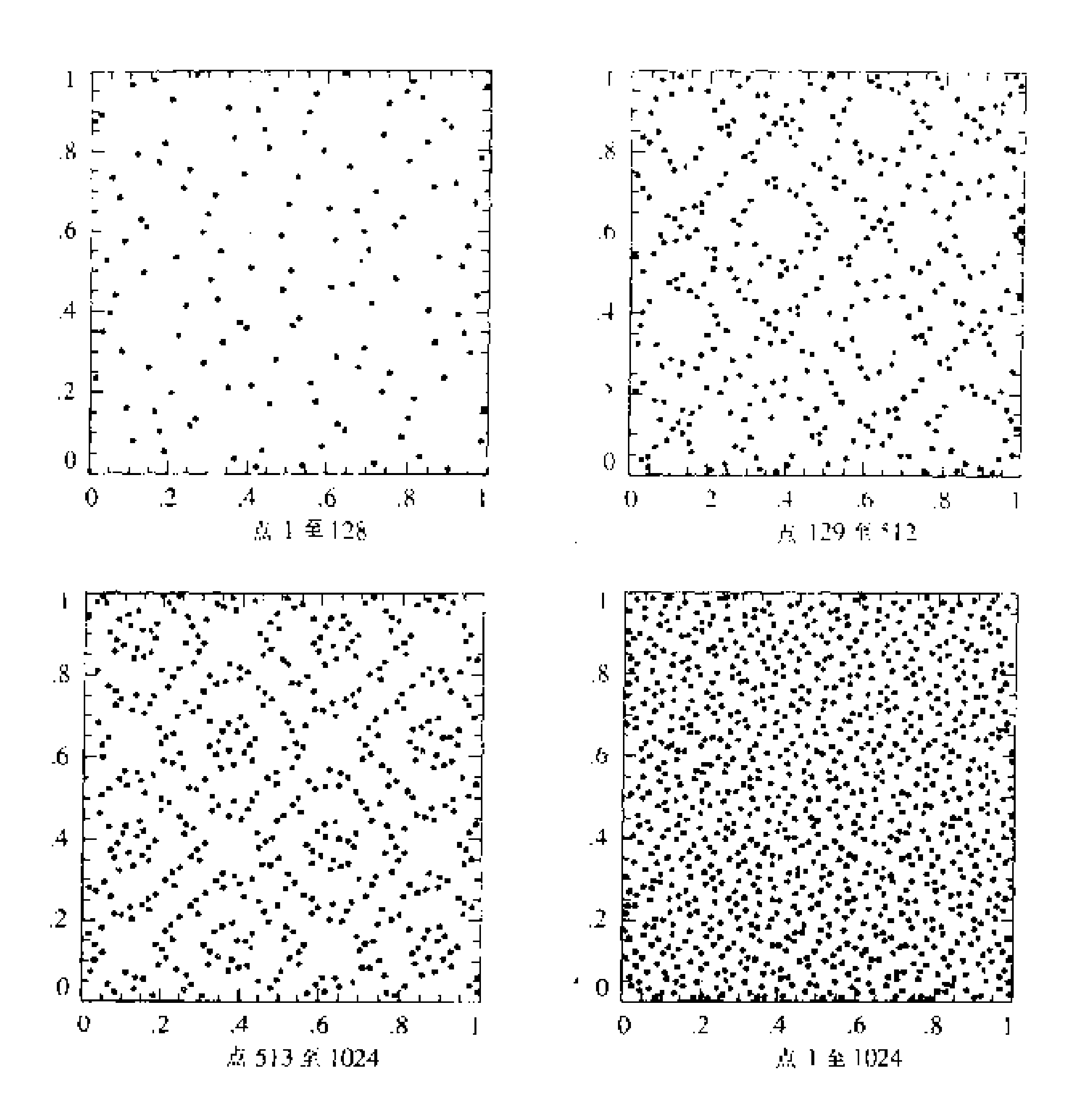
Faure,Sobol,Niederreiter 等人提出过产生准随机序列的其他方法。Bratley 与 Fox[2] 给出了一个很好评论及论述。并讨论了由 Antonov-Saleev[4] 提供的 Sobol[3] 序列的特别有效的变量。我们现在要讨论的就是 Antonov-Saleev 变量的实施。

从一组特殊的 $w$ 个二进制尾数 $V_i$,$i=1,2,\cdots,w$ 中,Sobol序列产生0与1之间的数,直接作为长度为 $w$ 位的二进制尾数,这叫做**方向数**,在 Sobol 原来的方法中,第 $j$ 个数 $X_j$ 是由满足条件的一些数组 $V_i$(逐位异或)产生的,其条件是 $V_i$ 的 $i$ 要满足"$j$ 的第 $i$ 位为非零"。换言之,当 $j$ 增加时,$V_i$ 中的不同的 I,以不同的时间进入或消失在 $X_j$ 中,这就使 $V_1$ 交替地出现和消失,变化得非常快,而 $V_k$ 从出现到消失(或反之)要每 $2^{k-1}$ 步产生一次。

Antonov 和 Saleev 的贡献是用整数 $j$ 的比特位替换方向数,人们也可用 $j$ 的格雷码 $G(j)$(Gray Code)的比特位(格雷码的简要论述参见第20.2节)。

现在，$G(j)$与$G(j+1)$正好只有一位比特位不同，即$j$的二进制表示的，最右边零位的位置（如占$i$，在$j$中加上一个前置零）。这样，第$j+1$个Sobol-Antonov Saleev数，可以从第$j$个数与单个$V_i$"异或"来得到，即$j$中最右边零位$i$。我们将看到，这使得序列的计算非常有效。

图7.7.1画出了二维Sobol序列产生的前1024个点，可以看出从相继点能"知道"前边留下的空隙，并分层地填充这些空隙。




该序列由数论生成，而不是随机的，故在任何阶段，相继点都"知道"应该如何填充到前面已有分布的空隙中

图7.7.1　二维索博(Sobol)序列的前1024个点

现在讨论方向数$V_i$的怎样产生的。许多优秀的数学家正在从事这项工作，所以，我们只是做一点简介以满足我们说明问题：每一个不同的Sobol序列（或$n$维序列的分量）是基于整数模2（Modulo 2）的各种本原多项式，即该多项式系数为0或1，并且不能再提出因子（用模2整数运算）成为较低次多项式。（模2的本原多项式在第7.4节中用过，在第20.3节中还将进一步讨论）。假设$P$是这样一个$q$次多项式

$$P = x^q + a_1x^{q-1} + a_2x^{q-2} + \cdots + a_{q-1}x + 1 \qquad (7.7.1)$$

用$q$项递推关系定义一个整数序列$M_i$，

$$M_i = 2a_1M_{i-1} \oplus 2^2a_2M_{i-2} \oplus \cdots \oplus 2^{q-1}a_{q-1}M_{i-q+1} \oplus (2^qM_{i-q} \oplus M_{i-q}) \qquad (7.7.2)$$

其中逐位异或以$\oplus$表示。这个递推关系的开始值$M_1, \cdots, M_q$是可以分别小于$2, \cdots, 2^q$的任意奇整数。由方向

数 $V_i$ 为：

$$V_i = M_i / 2^i \qquad i = 1, \ldots, w \tag{7.7.3}$$

表7.7.1列出了所有级次 $q<10$ 的模2本原多项式。由于各项系数为0或1，且 $x^q$ 和1项的系数总是1，故用中间各项的系数作为一个二进制数的各比特位，由此来表示多项式是很方便的（$x$ 的幂次越高有效位数越多）。该表使用此约定。

**表7.7.1　模2本原多项式**

| 阶次 | 模 2* 本原多项式 |
|---|---|
| 1 | 0 (i.e., $x+1$) |
| 2 | 1 (i.e., $x^2+x+1$) |
| 3 | 1, 2 (i.e., $x^3+x+1$ and $x^3+x^2+1$) |
| 4 | 1, 4 (i.e., $x^4+x+1$ and $x^4+x^3+1$) |
| 5 | 2, 4, 7, 11, 13, 14 |
| 6 | 1, 13, 16, 19, 22, 25 |
| 7 | 1, 4, 7, 8, 14, 19, 21, 28, 31, 32, 37, 41, 42, 50, 55, 56, 59, 62 |
| 8 | 14, 21, 22, 38, 47, 49, 50, 52, 56, 67, 70, 84, 97, 103, 115, 122 |
| 9 | 8, 13, 16, 22, 25, 44, 47, 52, 55, 59, 62, 67, 74, 81, 82, 87, 91, 94, 103, 104, 109, 122, 124, 137, 138, 143, 145, 152, 157, 167, 173, 176, 181, 182, 185, 191, 194, 199, 218, 220, 227, 229, 230, 234, 236, 241, 244, 253 |
| 10 | 4, 13, 19, 22, 50, 55, 64, 69, 98, 107, 115, 121, 127, 134, 140, 145, 152, 158, 161, 171, 181, 194, 199, 203, 208, 227, 242, 251, 253, 265, 266, 274, 283, 289, 295, 301, 316, 319, 324, 346, 352, 361, 367, 382, 395, 398, 400, 412, 419, 422, 426, 428, 433, 446, 454, 457, 472, 493, 505, 508 |

*用十进制整数表示内部比特位（即忽略最高次的比特位和单位1的比特位）

**表7.7.2　用于 Sobseq 的初始值**

| 阶次 | 多项式 | 初始值 | | | |
|---|---|---|---|---|---|
| 1 | 0 | 1 | (3) | (5) | (15)… |
| 2 | 1 | 1 | 1 | (7) | (11)… |
| 3 | 1 | 1 | 3 | 7 | (5)… |
| 3 | 2 | 1 | 3 | 3 | (15)… |
| 4 | 1 | 1 | 1 | 3 | 13… |
| 4 | 4 | 1 | 1 | 5 | 9… |

括号中的值不能随便使用，而要用这一阶次的递推关系得出

现在讨论 Sobol 序列的实施过程。连续调用 **Sobseq** 函数(在初始调用后)，返回一个 $n$ 维 Sobol 序列的连续点，它是基于表7.7.1中前 $n$ 个本原多项，正如已给出的程序中初始化 $n$，使其最大为 6 维，字长 $w$ 为 30 位。这些参数可通过 MAXBIT($\equiv w$)和 MAXDIM 来改变，还可以通过在数组 ip(表中的本原多项式)，mdeg(它的阶次)，及 iv(递推的初始数据，方程(7.7.2))中添加更多的初始数据来改变。在上页表7.7.2中，表明了程序中使用的初始数据。

```
#include "nrutil.h"
#define MAXBIT 30
#define MAXDIM 6

void sobseq(int *n, float x[])
    当 n 为负值时，对 MAXDIM 每个不同的 Sobol 序列，内部初始化一组 MAXBIT 方向数。当 n 是正值(但≤
    MAXDIM)，以向量 x[1..n]返回这 n 维序列的下一个值。
{
    int j,k,l;
    unsigned long i,im,ipp;
    static float fac;
    static unsigned long in,ix[MAXDIM+1],*iu[MAXBIT+1];
    static unsigned long mdeg[MAXDIM+1]={0,1,2,3,3,4,4};
    static unsigned long ip[MAXDIM+1]={0,0,1,1,2,1,4};
    static unsigned long iv[MAXDIM*MAXBIT+1]={
        0,1,1,1,1,1,1,3,1,3,3,1,1,5,7,7,3,3,5,15,11,5,15,13,9};

    if (*n < 0) {                                 初始化，不返回向量
        for (j=1,k=0;j<=MAXBIT;j++,k+=MAXDIM) iu[j] = &iv[k];
        允许1D和2D访问！
        for (k=1;k<=MAXDIM;k++) {
            for (j=1;j<=mdeg[k];j++) iu[j][k] <<= (MAXBIT-j);
            Stored values only require normalization.
            for (j=mdeg[k]+1;j<=MAXBIT;j++) {     用递推得到另一值
                ipp=ip[k];
                i=iu[j-mdeg[k]][k];
                i ^= (i >> mdeg[k]);
                for (l=mdeg[k]-1;l>=1;l--) {
                    if (ipp & 1) i ^= iu[j-l][k];
                    ipp >>= 1;
                }
                iu[j][k]=i;
            }
        }
        fac=1.0/(1L << MAXBIT);
        in=0;
    } else {                                      计算序列中的下一向量
        im=in;
        for (j=1;j<=MAXBIT;j++) {                 找出最右边的零比特位
            if (!(im & 1)) break;
            im >>= 1;
        }
        if (j > MAXBIT) nrerror("MAXBIT too small in sobseq");
        im=(j-1)*MAXDIM;
        for (k=1;k<=IMIN(*n,MAXDIM);k++) {        异或适当的方向数成为向量的每个分量，
            ix[k] ^= iv[im+k];                        并且转换成浮点数
            x[k]=ix[k]*fac;
        }
        in++;                                     数增1
    }
}
```

Sobol 序列好在什么地方?对 $n$ 维平滑函数的蒙特卡罗积分，其回答是，对随机取样数 $N$ 相对误差以

$(\ln N)^3/N$ 的方式减小,也就是几乎和 $1/N$ 一样快。作为示例,设一个积分函数 $f$,它在三维空间的圆环内是非零的。如果圆环的主半径是 $R_0$。辅助半径的坐标 $r$ 定义为

$$r = ([(x^2 + y^2)^{1/2} - R_0]^2 + z^2)^{1/2} \tag{7.7.4}$$

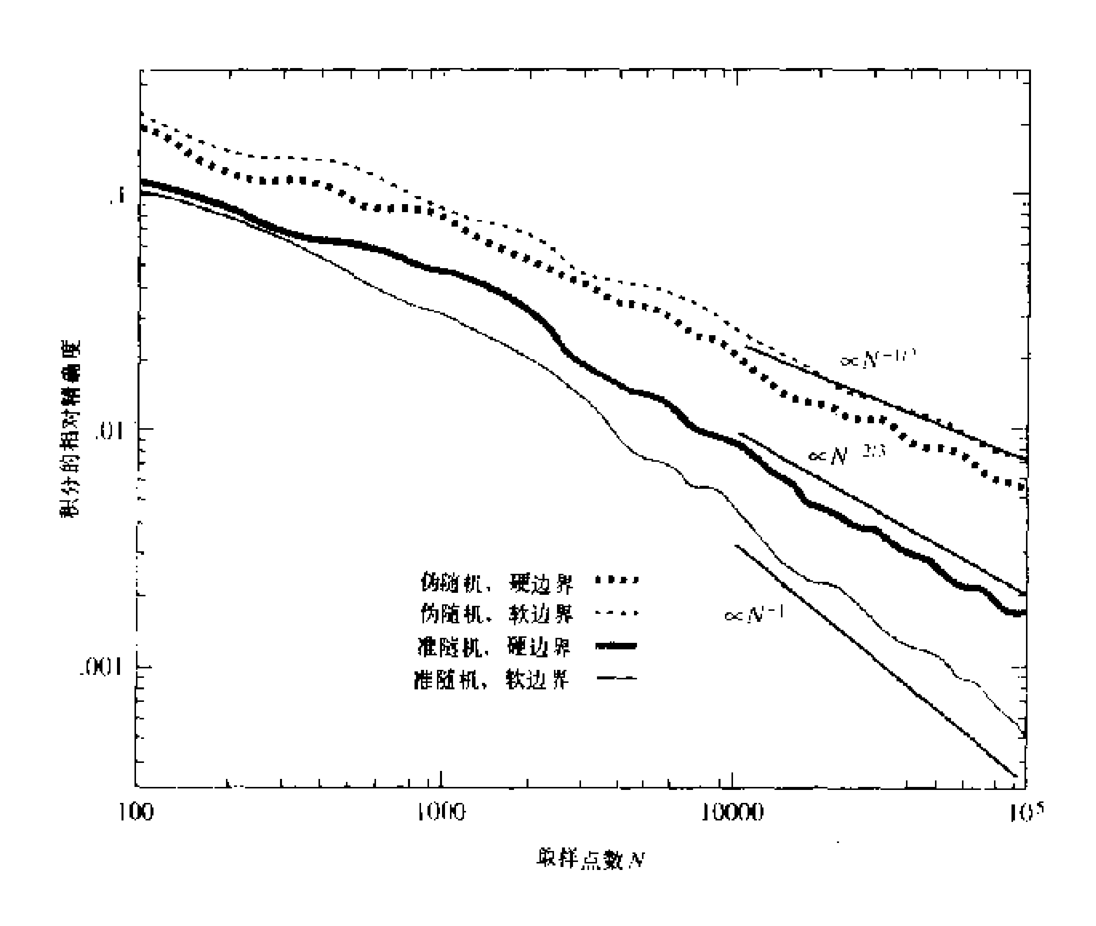
函数

$$f(x,y,z) = \begin{cases} 1 - \cos\left(\dfrac{\pi r^2}{a^2}\right) & r < r_0 \\ 0 & r \geq r_0 \end{cases} \tag{7.7.5}$$

可在圆柱形坐标系得解析积分,有

$$\iiint dx\,dy\,dz\,f(x,y,z) = 2\pi^2 a^2 R_0 \tag{7.7.6}$$

方程(7.7.4)在区间 $-1<x,y,z<1$ 内均匀取样,并以参数 $R_0=0.6$,$r_0=0.3$ 和由 **Sobseq** 程序产生三个不相关的随机点和 Sobol 序列,做相继100次蒙特卡罗积分,结果示于图7.7.2。图中曲线表示100次积分的均方根平均误差与取样点数之间的函数关系(对作一单个积分,因有漂移误差从正值到负值,或相反,所以相对误差的对数图没有多少信息)。图中细短划曲线对应于不相关的随机取样点,并显示了常用的 $N^{-1/2}$ 渐近线。细实曲线显示了 Sobol 序列的结果。表达式 $(\ln N)^3/N$ 中的对数项已表示在曲线的曲率中,而渐近线 $N^{-1}$ 是明确的。




图中对应于两个不同被积分函数与两个不同选择随机点的方法,准随机 Sobol 序列收敛得比传统的伪随机序列快得多。对准随机取样来说,平滑的被积函数(软边界)将比阶跃跳变的被积函数(硬边界)好得多,曲线显示了100次试验的均方根平均值。

图7.7.2　蒙特卡罗积分的相对精确度和取样点数的函数关系

为了理解图7.7.2的重要性,假设要求函数 $f$ 的蒙特卡罗积分的精确度在1%以内。Sobol 序列要进行

几千次取样方可达到这个精确度，而伪随机取样几乎需要十万次。对于更高精确度的要求，这个比率还会更高。

与上不同，当被积函数在取样区域内有硬的边界（不连续性），情况就不太理想，例如，函数值在圆环内取 1，外面取 0：

$$f(x,y,z)=\begin{cases}1 & r<r_0\\ 0 & r\geq r_0\end{cases} \tag{7.7.7}$$

其中 $r$ 由式(7.7.4)定义。恰巧，这个函数有式(7.7.6)函数相同的解析积分，即 $2\pi^2a^2R$。

细致地分层次的 Sobol 序列是基于一组笛卡尔栅格上，但（圆环面）边界与这些栅格没有明确关系。结果是在圆环表面薄层上有量级为 $N^{2/3}$ 的取样点，它无论来自圆环面内，还是圆环外大致都是随机的。对这薄层应用平方根定律得到总数的 $N^{1/3}$ 偏离，或蒙特卡罗积分 $N^{-2/3}$ 相对误差。图7.7.2的粗实线证实了这个特性。图7.7.2中粗的短划线是函数式(7.7.7)用独立随机点的积分结果，尽管 Sobol 序列的优点不像在平滑函数时那么突出，但它的精确度仍然适中，比如1%时，平滑函数的情况要优越一个因子(～5)，当精确度提高时，这个因子还将增大。

要知道，我们提供的 Sobol 程序产生的序列都是从起点开始的，若要满足从某点而不从起点开始产生序列，这是很容易做到的。一旦方向数 iv 初始化了，第 $j$ 个点能够直接由 $j$ 的格雷码中的非零位所对应的方向数进行异或得到。正如上面描述的那样。

### 7.7.1 拉丁(Latin)超立方

我们在这里简要叙述一下**拉丁**(Latin)**平方**或**拉丁超立方**取样的不相关技术，当必须在 $N$ 维空间中极稀疏地取样时(取样点为 $M$)，这种技术是很有用的。例如，当要试验汽车的抗撞性能，它是四个不同的设计参数的联立函数，但从经济考虑只有三部汽车可供试验(这是不是一个好的方案倒不是关键，而是如何尽量利用现有的条件)。

设计思想是，将每一个设计参数(维数)分为 $M$ 段，故整个空间分成 $M^N$ 个小单元(每一维选择的段 $M$，可以相等或不相等)。例如，三部汽车四个参数就共有3×3×3×3＝81个小单元。

其次，按下列算法选择 $M$ 个小单元包含的取样点：随机地选择 $M^N$ 个小单元中一个，作为第一点。现除去与第一点有某一相同参数的所有小单元(也就是，划去与第一点在同一行或同一列的所有小单元)，还剩下 $(M-1)^N$ 个小单元。再随机地选取其中之一，并划去新的行与列。一直重复这个步骤直到只剩下一个小单元，它包含最后一个样点。

这种结构的特点是，每一个设计参数都将在它的每一个子区域里经过测试。如果测试时系统响应由其一个设计参数所控制，则这个参数会在这个取样技术中被发现。另一方面，如果各参数之间有强烈的相互影响，拉丁超立方将失去它的优势，所以使用时要小心。

参考文献与进一步阅读：


Halton, J. H. 1960, *Numerische Mathematik*, vol. 2, pp. 84-90. [1]

Bratley P., and Fox, B. L. 1988, *ACM Transactions on Mathematical Software*, vol. 14, pp. 88～100. [2]

Lambert, J. P. 1988, in *Numerical Mathematics - Singapore* 1988, ISNM vol. 86, R. P. Agarwal, Y. M. Chow, and S. J. Wilson, eds. (Basel: Birkhauser), pp. 273～284.

Niederreiter, H. 1988, in *Numerical Integration III*, ISNM vol. 85, H. Brass and G. Hammerlin, eds. (Basel: Birkhauser), pp. 157～171.

Sobol', I.M. 1967, *USSR Computational Mathematics and Mathematical Physics* vol. 7, no. 4, pp. 86~112. [3]

Antonov, I. A., and Saleev, V. M 1979, USSR *Computational Mathematics and Mathematical Physics*, vol. 19, no. 1, pp. 252~256. [4]

## 7.8 自适应及递归蒙特卡罗方法

这一节讨论蒙特卡罗积分更先进的技术。作为应用这些技术的例子，我们举两个很不相同，但很有技巧的多维蒙特卡罗程序：**vegas**[1,2]与 **miser**[3]。我们讨论的技术都属于**方差退化**的一般程序(第7.6节)，但是又很有特色。

### 7.8.1 重要取样

**重要取样**的应用已隐含在方程(7.6.6)与(7.6.7)中。我们现在采用比较正规的方法讨论它。设被积函数 $f$ 可以写为两个函数的乘积，一个函数 $h$ 几乎是常数，另一个函数 $g$ 是正值函数，则它对多维体积 $V$ 的积分是

$$\int f\,dV = \int (f/g)g\,dV = \int hg\,dV \tag{7.8.1}$$

根据方程(7.6.7)，我们可以将方程(7.8.1)进行变量变换成 $G$，而 $G$ 是 $g$ 的不定积分。这使 $g\,dV$ 成为一个全微分。然后，我们应用蒙特卡罗积分的基本定理——方程(7.6.1)进行处理。

方程(7.8.1)更一般的理解是，积分 $f$ 可以由取样 $h$ 代替，但此时不是均匀概率密度 $dV$，而是非均匀密度 $g\,dV$，前面第一种理解是第二种理解的特殊情形，产生非均匀取样 $g\,dV$ 的方法就是通过使用不定积分 $G$ 的变换方法(见第7.2节)。

更直接地，可以把基本定理(7.6.1)推广到非均匀取样的情形：设在体积 $V$ 中选取具有概率密度 $p$ 的取样点 $x_i$，它满足

$$\int p\,dV = 1 \tag{7.8.2}$$

推广后的基本定理是，用 $N$ 个取样点 $x_1,\ldots,x_N$，任意函数 $f$ 的积分是估算值 $I$

$$I \equiv \int f\,dV = \int \frac{f}{p}\,p\,dV \approx \left\langle\frac{f}{p}\right\rangle \pm \sqrt{\frac{\langle f^2/p^2\rangle - \langle f/p\rangle^2}{N}} \tag{7.8.3}$$

这里角括号表示 $N$ 个点的算术平均值，精确的如式(7.6.2)。式中的正负项是误差估计的标准偏差，如式(7.6.1)，要注意，式(7.6.1)是式(7.8.3)在 $p=$常数$=1/V$ 的特殊情形。

取样密度 $p$ 的最佳选择是什么呢？我们已直观地看到，就是使 $h=f/p$ 尽可能接近常数。我们可以把注意力集中在式(7.8.3)根号内的分子上，即每个取样点的方差。两个角括号都是积分的蒙特卡罗估计量，所以可以写成

$$S \equiv \left\langle\frac{f^2}{p^2}\right\rangle - \left\langle\frac{f}{p}\right\rangle^2 \approx \int \frac{f^2}{p^2}\,p\,dV - \left[\int \frac{f}{p}\,p\,dV\right]^2 = \int \frac{f^2}{p}\,dV - \left[\int f\,dV\right]^2 \tag{7.8.4}$$

我们现在用函数变分，求式(7.8.2)限制的条件下，$p$ 的最佳值

$$0 = \frac{\delta}{\delta p}\left(\int \frac{f^2}{p}\,dV - \left[\int f\,dV\right]^2 + \lambda\int p\,dV\right) \tag{7.8.5}$$

其中 $\lambda$ 是拉格朗日乘子。注意中间项并不依赖 $p$，这变分(来自积分内部)给出 $0=-f^2/p^2+\lambda$ 或

$$p = \frac{|f|}{\sqrt{\lambda}} = \frac{|f|}{\int |f|\,dV} \tag{7.8.6}$$

这里的 $\lambda$ 是受式(7.8.2)限制的。

如果在积分区域 $f$ 不变号，如果有计算取样结果的特殊方法，则可得出明显的结果，$p$ 的最佳值正比

于$|f|$。于是方差值将减小到0。虽不太直观，但的确是：即使$f$要变号，最佳值也将有$p\propto|f|$。在这种情形下，每取样点的方差(式(7.8.4)与(7.8.6))是

$$S - S_{最佳} = \left(\int |f|\, dV\right)^2 - \left(\int f\, dV\right)^2 \tag{7.8.7}$$

有意思的是，在被积函数中，可以加一个常数而保持它不变号，因为这仅使积分变化一个已知量：常数乘$V$。$p$的最佳选择总是给出零方差，积分完全精确！对这个看来是似而非的问题(已在第7.8节末尾叙述过)的回答是这样的：要确切知道式(7.8.6)中$p$，就要确切知道积分$\int f\, dV$，这就等价于确切知道待计算的积分！

如果函数$f$在体积$V$的大部分区域有一已知常数值，一个好的方法是，再加一常数使那个常数值为零。这样做大概只是一个粗略不成熟想法，重要取样所能达到的精确度，不取决于式(7.8.7)取多小，而实际上取决于对可执行的$p$，式(7.8.4)取多小。

## 7.8.2 分层取样

分层取样的思路完全不同于重要抽样。我们将符号扩充一下，令$\langle\langle f\rangle\rangle$表示函数$f$对体积$V$的真实平均(即积分被$V$除)，$\langle f\rangle$如前一样是对取样点(均匀取样)的算术平均：

$$\langle\langle f\rangle\rangle \equiv \frac{1}{V}\int f\, dV \qquad \langle f\rangle \equiv \frac{1}{N}\sum_i f(x_i) \tag{7.8.8}$$

估计值的方差$\mathrm{Var}(\langle f\rangle)$是蒙特卡罗积分误差平方的量度，它由关系式

$$\mathrm{Var}(\langle f\rangle) = \frac{\mathrm{Var}(f)}{N} \tag{7.8.9}$$

确定(对比式(7.6.1))，与函数的方差$\mathrm{Var}(f)\equiv\langle\langle f^2\rangle\rangle-\langle\langle f\rangle\rangle^2$有关。

设将体积$V$分割成两个相等的、不相交的子体积$a$和$b$，在每一子体积中取$N/2$个样点。不同于式(7.8.8)，$\langle\langle f\rangle\rangle$的估计值$\langle f\rangle'$表示为

$$\langle f\rangle' \equiv \frac{1}{2}(\langle f\rangle_a + \langle f\rangle_b) \tag{7.8.10}$$

换言之，它是在两个半区域内的取样平均的均值。估计值(7.8.10)的方差是

$$\begin{aligned}\mathrm{Var}(\langle f\rangle') &= \frac{1}{4}[\mathrm{Var}(\langle f\rangle_a) + \mathrm{Var}(\langle f\rangle_b)]\\ &= \frac{1}{4}\left[\frac{\mathrm{Var}_a(f)}{N/2} + \frac{\mathrm{Var}_b(f)}{N/2}\right]\\ &= \frac{1}{2N}[\mathrm{Var}_a(f) + \mathrm{Var}_b(f)]\end{aligned} \tag{7.8.11}$$

这里$\mathrm{Var}_a(f)$表示$f$在子体积$a$中的方差$\langle\langle f^2\rangle\rangle_a-\langle\langle f\rangle\rangle_a^2$；对$b$也有对应量。

从已有的定义(在物理上，这个有关组合二阶矩的公式叫做"平行移轴定理"。)不难证明关系式：

$$\mathrm{Var}(f) = \frac{1}{2}[\mathrm{Var}_a(f) + \mathrm{Var}_b(f)] + \frac{1}{4}(\langle\langle f\rangle\rangle_a - \langle\langle f\rangle\rangle_b)^2 \tag{7.8.12}$$

比较公式(7.8.9)、(7.8.11)和(7.8.12)可以看出，分层(分为两个子体积)取样法给出的方差——当$\langle\langle f\rangle\rangle_a$与$\langle\langle f\rangle\rangle_b$不同时，它是较小的，决不比简单蒙特卡罗情形的大。

我们还没有利用在两个子体积中，取不同数量的点数的可能性，如在子体积$a$中取$N_a$点，而在子体积$b$中取$N_b\equiv N-N_a$点。现在就来研究这个问题。估计值的方差为

$$\mathrm{Var}(\langle f\rangle') = \frac{1}{4}\left[\frac{\mathrm{Var}_a(f)}{N_a} + \frac{\mathrm{Var}_b(f)}{N-N_a}\right] \tag{7.8.13}$$

很容易证明它在

$$\frac{N_a}{N} = \frac{\sigma_a}{\sigma_a+\sigma_b} \tag{7.8.14}$$

时取极小值。

这里我们选用了简化标记 $\sigma_a \equiv [\mathrm{Var}_a(f)]^{1/2}$，$b$ 亦类似。如果 $N_a$ 满足方程(7.8.11)，则方程(7.8.13)简化为：

$$\mathrm{Var}(\langle f\rangle') = \frac{(\sigma_a + \sigma_b)^2}{4N} \tag{7.8.14}$$

如果 $\mathrm{Var}(f) = \mathrm{Var}_a(f) = \mathrm{Var}_b(f)$，方程(7.8.13)简化为(7.8.9)，在这种情况下分层取样没起作用。

概括归纳上面的标准方法是，将体积 $V$ 分割成多于两个相等的子区域。在各区域间分配取样点数的最佳方案是，令每一区域 $j$ 的点数正比于 $\sigma_j$(即在这子区域中函数 $f$ 的方差的平方根)。但在高维空间中(如 $d \gtrsim 4$)，这实际上并不是很有用。沿每一维将体积分割为 $K$ 部分，其意味着有 $K^d$ 个子体积，在必须估计全部对应的 $\sigma_j$ 时，一般这个数目太大了。

## 7.8.3 混合策略

初看起来，重要取样与分层取样是互不相容的，前者在被积函数值 $|f|$ 最大的地方集中取样点，而后者集中在 f 的方差最大的地方。为什么二者都可取呢？

其回答是，(也像生活中的一些事)它们取决于你知道什么和你知道多少。重要取样依赖于已经掌握的积分近似值，故能够产生具有所期望的概率密度 $p$ 的随机点 $x_i$。由于 $p$ 在一定程度上不理想，所以只能使误差仅以 $N^{-1/2}$ 减小。如果在被积函数 $f$ 急剧变化的区域，$p$ 很不理想，事情就会很糟，因为，取样函数 $h = f/p$ 会产生很大的方差。重要取样是研究光滑取样函数 $h$ 的值，只有满足这些条件才是有效的。

相反，分层取样不需要知道 $f$ 的任何信息。分层取样是研究消除各子区域点数的起伏，而不是研究各点平滑的数值。最简单的分层策略：是分割 $V$ 为相等的 $N$ 个子区域，并在每一个子区域中随机地选取一个点，这个方法的误差近似按 $N^{-1}$ 下降，比 $N^{-1/2}$ 快得多。(第7.7节中的准随机数是消除点密度起伏的另一个方法，它近似给出与"盲目"分层取样一样好的结果。)

然而，"近似"是一个很重要的提示：例如，假设被积函数除在某一个子区域外都可以忽略，则在这单个子区域内取样积分的结果完全无用。粗略可知，重要取样允许在这子区域中取很多点，这要比仔细分层取样好得多。

分层取样还是有它的独到之处，只要能估计出方差值，就能按照式(7.8.14)及其推论在不同子区域取不同数目的点，同时能找到将一区域划分为适当数目的子区域(注意，不是多维数 $d$ 的 $K^d$ 个)的方法，每一子区域的函数方差将比总体积中的方差显著减少，要做到这些，需要掌握函数 $f$ 的许多信息，虽然，重要取样也需要掌握 $f$ 的各种信息。

实际上，重要取样与分层取样，即使不是在很多方面，但也是在许多重要的方面是可兼容的。许多被积函数 $f$ 的值，除了称为"作用区"的小部分体积外，在 $V$ 中处处都是很小，而在这些作用区，$f$ 的数值与标准偏差 $\sigma = [\mathrm{Var}(f)]^{1/2}$ 的大小是可比较的，所以，这两个技术将给出大致相同的点密度。在较复杂的执行过程中，可能要把这两种技术"嵌套"起来，例如，基于粗栅格上的重要取样后，可以在每一栅格中，再采用分层取样。

## 7.8.4 自适应蒙特卡罗：VEGAS

Peter Lepage[1,2] 发明的 VEGAS 算法广泛应用于多维积分，基本粒子物理学中常用到。VEGAS 基本上采用重要取样，但若维数 $d$ 足够小时，可以避免 $K^d$ 的迅速增长，有时也采用分层取样(特别是如果 $(K/2)^d < N/2$，$N$ 为取样点数)。VEGAS 中重要取样的基本方法是，自适应地构造一个多维加权函数 $g$，它是可分解的，

$$p \propto g(x, y, z, \ldots) = g_x(x) g_y(y) g_z(z) \ldots \tag{7.8.16}$$

这样一个函数有两种方法避免 $K^d$ 的迅速增长：(1)可以把作为 $d$ 个分割的一维函数存储在计算机中，每一函数由 $K$ 个列表值定义，即用 $K \times d$ 代替 $K^d$。(2)对 $d$ 个一维函数，以概率密度进行连续取样，而得到坐标

向量分量 $(x, y, z, \cdots)$。

最佳可分解加权函数可写为[1]

$$g_x(x) \propto \left[\int dy \int dz \ldots \frac{f^2(x,y,z,\ldots)}{g_y(y)g_z(z)\ldots}\right]^{1/2} \tag{7.8.17}$$

(对 $y$,$z$ 也有类似方程)。此式在一维时,退化为 $g \propto |f|$。式(7.8.9)、式(7.8.17)显然提示了 VEGAS 的自适应策略:给出一组 $g$ 函数(如,初始可以都取为常数),对函数 $f$ 取样,不仅累积积分所有的估值,还累积式(7.8.17)右边的 $Kd$ 估计值($d$ 维的每一独立自变量分为 $K$ 段),由此决定下一迭代过程改进的 $g$ 函数。

若被积函数 $f$ 局限于 $d$ 维空间的一个或少数几个区域,则当坐标值为这些区域在该坐标轴的投影时,这些加权函数 $g$ 很快变大。蒙特卡罗积分的精确度将显著提高,超过简单蒙特卡罗的精确度。

VEGAS 的特点是显见的:函数 f 在个别坐标方向投影是均匀的,VEGAS 在这些方向上的取样点不集中。例如,VEGAS 的最坏情况是被积函数集中在全体对角线上,如,从(0,0,0,...)到(1,1,1,...)。因为这种几何形状是完全不可分割的,VEGAS 完全体现不出优点。更一般地,若被积函数集中在一维(或更高维)曲线上(或超曲面上),VEGAS 算法也许根本不能使用,除非刚好处于坐标方向。

下面列出的程序 Vegas 基本上是 Lepage 的标准版本,为适应我们的需要做了很少的修改。(感谢 Lepage 允许我们在这里引用些程序)。为了与 VEGAS 算法的其他版本在计算上相协调,我们保留了原来的变量名称。参数 NDMX 就是我们称作 K 的变量名,是沿每一轴的增量最大值;MXDIM 是 $d$ 的最大值;其他参数待注释中说明。

Vegas 程序执行 $m$=itmx 次对所求积分进行统计独立的计算,每一次具有 N=ncall 个函数计算。它们虽然是统计不相关的,但这些迭代是互相支持的,因为每一次迭代都用来为下一次迭代改进取样栅格。所有迭代结果,组合得到最佳答案及其估计误差,如下式

$$I_{最佳} = \sum_{i=1}^{m} \frac{I_i}{\sigma_i^2} \bigg/ \sum_{i=1}^{m} \frac{1}{\sigma_i^2} \qquad \sigma_{最佳} = \left(\sum_{i=1}^{m} \frac{1}{\sigma_i^2}\right)^{-1/2} \tag{7.8.18}$$

返回这个参量

$$\chi^2/m = \frac{1}{m-1}\sum_{i=1}^{m} \frac{(I_i - I_{最佳})^2}{\sigma_i^2} \tag{7.8.19}$$

如果它远远大于1,则迭代结果是统计不相容的,并答案是值得怀疑的。

输入标记符 init 使用很方便,可以调用 init=0, ncall=1000, itmx=5;然后调用 init=1, ncall=100000, itmx=1。执行结果可形成小取样数的 5 次迭代的取样栅格,然后在最佳栅格上做一个高精确度积分。

用户提供的被积函数 fxn 除所期望的求值点 x 外,还有一自变量 wgt,在多数应用场合可忽略函数内的 wgt。但有时可能要对一些附加函数或和主函数 $f$ 相一致的函数进行积分,如某些函数 $g$ 的积分可以估值

$$I_g = \sum_i w_i g(\mathbf{x}) \tag{7.8.20}$$

这里 $w_i$ 与 $\mathbf{x}$ 分别是自变量 wgt 与 x。在函数 fxn 内部直接累加求和,并通过全程变量将此求和结果返回到用户的主程序。当然,由于取样将对 $f$ 优化,所以 $g(\mathbf{x})$ 最好是与主函数 $f$ 有某种程度的类似。

```
#include <stdio.h>
#include <math.h>
#include "nrutil.h"
#define ALPH 1.5
#define NDMX 50
#define MXDIM 10
#define TINY 1.0e-30

extern long idum;                    在主程序中对随机数初始化
```

void vegas(float regn[], int ndim, float (*fxn)(float [], float), int init, unsigned long ncall, int itmx, int nprn, float *tgral, float *sd, float *chi2a)

此程序实现对用户提供的 ndim 维函数 fxn 在以 regn[1..2*ndim] 标志的长方体内进行蒙特卡罗积分。这个 regn[1..2*ndim] 向量由 ndim 的“较低左边”座标，其后跟随 ndim 的“较高右边座标而组成。积分包括 itmx 次迭代，每次近似 ncall 次调用函数。每次迭代后栅格被精练；大于5或10次迭代是相当有用的。输入标志 init 表示这调用是新的开始，还是为附加迭代的子调用（见下面注释）。输入标志 nprn（通常为0）是控制诊断量的输出。返回的答案是 tgral（积分的最佳估值），sd（它的标准差）以及 chi2a（每自由度的 $\chi^2$，一个是否一致的结果已获取的标记。）详情见前面正文。

```
{
    float ran2(long *idum);
    void rebin(float rc, int nd, float r[], float xin[], float xi[]);
    static int i,it,j,k,mds,nd,ndo,ng,npg,ia[MXDIM+1],kg[MXDIM+1];
    static float calls,dv2g,dxg,f,f2,f2b,fb,rc,ti,tsi,wgt,xjac,xn,xnd,xo;
    static float d[NDMX+1][MXDIM+1],di[NDMX+1][MXDIM+1],dt[MXDIM+1],
        dx[MXDIM+1], r[NDMX+1],x[MXDIM+1],xi[MXDIM+1][NDMX+1],xin[NDMX+1];
    static double schi,si,swgt;
    较好设为静态、允许重新开始   allowing restarts.

    if (init <= 0) {                    标准输入、在冷启动时输入
        mds=ndo=1;                      改变到mds=0使不能分层取样，即仅用重要取样．
        for (j=1;j<=ndim;j++) xi[j][1]=1.0;
    }
    if (init <= 1) si=swgt=schi=0.0;    这里送入以继承前次调用的栅格，但不送入它结果
                                        这里送入以继承前次调用的栅格和它结果
    if (init <= 2) {
        nd=NDMX;
        ng=1;
        if (mds) {                      为分层设置循环
            ng=(int)pow(ncall/2.0+0.25,1.0/ndim);
            mds=1;
            if ((2*ng-NDMX) >= 0) {
                mds = -1;
                npg=ng/NDMX+1;
                nd=ng/npg;
                ng=npg*nd;
            }
        }
        for (k=1,i=1;i<=ndim;i++) k *= ng;
        npg=IMAX(ncall/k,2);
        calls=npg*k;
        dxg=1.0/ng;
        for (dv2g=1,i=1;i<=ndim;i++) dv2g *= dxg;
        dv2g=SQR(calls*dv2g)/npg/npg/(npg-1.0);
        xnd=nd;
        dxg *= xnd;
        xjac=1.0/calls;
        for (j=1;j<=ndim;j++) {
            dx[j]=regn[j+ndim]-regn[j];
            xjac *= dx[j];
        }
        if (nd != ndo) {                若需要反复量化分级
            for (i=1;i<=nd;i++) r[i]=1.0;
            for (j=1;j<=ndim;j++) rebin(ndo/xnd,nd,r,xin,xi[j]);
            ndo=nd;
        }
        if (nprn >= 0) {
            printf("%s: ndim= %3d ncall= %8.0f\n",
             " Input parameters for vegas",ndim,calls);
            printf("%28s it=%5d itmx=%5d\n"," ",it,itmx);
            printf("%28s nprn=%3d ALPH=%5.2f\n"," ",nprn,ALPH);
            printf("%28s mds=%3d nd=%4d\n"," ",mds,nd);
            for (j=1;j<=ndim;j++) {
                printf("%30s xl[%2d]= %11.4g xu[%2d]= %11.4g\n",
```

```
                " ",j,regn[j],j,regn[j+ndim]);
        }
    }
}
for (it=1;it<=itmx;it++) {
```
主要的迭代循环。这里能进入(init > 3)以完成附加的itmx次迭代，而所有其他参量不变

```
    ti=tsi=0.0;
    for (j=1;j<=ndim;j++) {
        kg[j]=1;
        for (i=1;i<=nd;i++) d[i][j]=di[i][j]=0.0;
    }
    for (;;) {
        fb=f2b=0.0;
        for (k=1;k<=npg;k++) {
            wgt=xjac;
            for (j=1;j<=ndim;j++) {
                xn=(kg[j]-ran2(&idum))*dxg+1.0;
                ia[j]=IMAX(IMIN((int)(xn),NDMX),1);
                if (ia[j] > 1) {
                    xo=xi[j][ia[j]]-xi[j][ia[j]-1];
                    rc=xi[j][ia[j]-1]+(xn-ia[j])*xo;
                } else {
                    xo=xi[j][ia[j]];
                    rc=(xn-ia[j])*xo;
                }
                x[j]=regn[j]+rc*dx[j];
                wgt *= xo*xnd;
            }
            f=wgt*(*fxn)(x,wgt);
            f2=f*f;
            fb += f;
            f2b += f2;
            for (j=1;j<=ndim;j++) {
                di[ia[j]][j] += f;
                if (mds >= 0) d[ia[j]][j] += f2;
            }
        }
        f2b=sqrt(f2b*npg);
        f2b=(f2b-fb)*(f2b+fb);
        if (f2b <= 0.0) f2b=TINY;
        ti += fb;
        tsi += f2b;
        if (mds < 0) {                使用分层取样
            for (j=1;j<=ndim;j++) d[ia[j]][j] += f2b;
        }
        for (k=ndim;k>=1;k--) {
            kg[k] %= ng;
            if (++kg[k] != 1) break;
        }
        if (k < 1) break;
    }
    tsi *= dv2g;                      为迭代计算最后结果
    wgt=1.0/tsi;
    si += wgt*ti;
    schi += wgt*ti*ti;
    swgt += wgt;
    *tgral=si/swgt;
    *chi2a=(schi-si*(*tgral))/(it-0.9999);
    if (*chi2a < 0.0) *chi2a = 0.0;
    *sd=sqrt(1.0/swgt);
    tsi=sqrt(tsi);
    if (nprn >= 0) {
        printf("%s %3d : integral = %14.7g +/- %9.2g\n",
         " iteration no.",it,ti,tsi);
        printf("%s integral =%14.7g+/-%9.2g chi**2/IT n = %9.2g\n",
            " all iterations: ",*tgral,*sd,*chi2a);
```

```
        if (nprn) {
            for (j=1;j<=ndim;j++) {
                printf(" DATA FOR axis %2d\n",j);
                printf("%6s%13s%11s%13s%11s%13s\n",
                 "X","delta i","X","delta i","X","delta i");
                for (i=1+nprn/2;i<=nd;i += nprn+2) {
                    printf("%8.5f%12.4g%12.5f%12.4g%12.5f%12.4g\n",
                        xi[j][i],di[i][j],xi[j][i+1],
                           di[i+1][j],xi[j][i+2],di[i+2][j]);
                }
            }
        }
    }
    for (j=1;j<=ndim;j++) {          精练栅格，查阅参考文献以便理解这些步骤的奥妙、
        xo=d[1][j];                      为避免快速、不稳定的变化，精练被阻止，并且
        xn=d[2][j];                      被指数ALPH压缩在范围内
        d[1][j]=(xo+xn)/2.0;
        dt[j]=d[1][j];
        for (i=2;i<nd;i++) {
            rc=xo+xn;
            xo=xn;
            xn=d[i+1][j];
            d[i][j] = (rc+xn)/3.0;
            dt[j] += d[i][j];
        }
        d[nd][j]=(xo+xn)/2.0;
        dt[j] += d[nd][j];
    }
    for (j=1;j<=ndim;j++) {
        rc=0.0;
        for (i=1;i<=nd;i++) {
            if (d[i][j] < TINY) d[i][j]=TINY;
            r[i]=pow((1.0-d[i][j]/dt[j])/
                (log(dt[j])-log(d[i][j])),ALPH);
            rc += r[i];
        }
        rebin(rc/xnd,nd,r,xin,xi[j]);
    }
  }
}


void rebin(float rc, int nd, float r[], float xin[], float xi[])
vegas所用的实用程序，把密度为xi的向量重新分成由向量r定义的新量级

{
    int i,k=0;
    float dr=0.0,xn=0.0,xo;

    for (i=1;i<nd;i++) {
        while (rc > dr) {
            dr += r[++k];
            xo=xn;
            xn=xi[k];
        }
        dr -= rc;
        xin[i]=xn-(xn-xo)*dr/r[k];
    }
    for (i=1;i<nd;i++) xi[i]=xin[i];
    xi[nd]=1.0;
}
```

### 7.8.5 递归分层取样

我们已经看到，分层取样存在的问题是不可避免 $K^d$ 的迅速增长，这显然是笛卡尔坐标 $d$ 维值所固有

的。一种称为**递归分层取样**[1]的技术试图解决这个问题，它不是沿所有 d 维，而是一次仅沿一维逐次将体积分半。对抽点应用逐次减小的子区域分半法的方程是(7.8.10)与(7.8.13)。

假设我们对函数 $f$ 定量地进行 $N$ 次计算，并求它在长方体 $R=(\mathbf{x}_a,\mathbf{x}_b)$ 内的中值 $\langle f\rangle'$（这样为区域由两个对顶角坐标向量表示）。首先，我们指定 $N$ 的一小部分 $p$ 来确定 $R$ 中 $f$ 的方差：在 $R$ 中均匀地抽样 $pN$ 个函数值并累加求和，对应于 $d$ 个不同的坐标方向（$R$ 能沿其方向分半），它将给出 $d$ 对不同的方差。换言之，在 $pN$ 个抽样中，在每一个基于 $R$ 可能的分半区域中我们估计 $\mathrm{Var}(f)$，

$$\begin{aligned} R_{ai} &\equiv (\mathbf{x}_a, \mathbf{x}_b - \frac{1}{2}\mathbf{e}_i \cdot (\mathbf{x}_b - \mathbf{x}_a)\mathbf{e}_i) \\ R_{bi} &= (\mathbf{x}_a + \frac{1}{2}\mathbf{e}_i \cdot (\mathbf{x}_b - \mathbf{x}_a)\mathbf{e}_i, \mathbf{x}_b) \end{aligned} \tag{7.8.21}$$

这里 $\mathbf{e}_i$ 是第 $i$ 个坐标方向的单位向量，$i=1,2,\ldots,d$。

其次，研究方差以找出分半的最佳维数 $i$。例如，我们可用方程(7.8.15)选择 $i$，以使在区域 $R_{ai}$ 和 $R_{bi}$ 中估计方差的平方根之和取最小值（将要说明，我们的实际做法稍有不同）。

第三，我们限定余下的 $(1-p)N$ 个函数在区域 $R_{ai}$ 与 $R_{bi}$ 之间计算求值。如果我们用方程(7.8.15)选择 $i$，则就应该按照方程(7.8.14)做这样的限定。

为估计 $f$ 的平均值，我们有两个长方体，每一个有自己的函数计算界限。我们的"RSS"算法，它本身是递归的：为计算每一区域的平均值，我们返回到式(7.8.21)上面的"首先..."句子重新开始（当然，如果此区域的点限定在某数以下，则我们常用简单蒙特卡罗法而不用连续递归）。

最后，我们综合这些方法，并用式(7.8.10)和式(7.8.11)的第一行，来估算两个子体积的方差。

这样以最简形式完成了 RSS 算法，在我们按"执行细节"的一般规程描述某些附加技巧之前，我们需要简要回顾一下式(7.8.13)至(7.8.15)，并推导出实际使用这些公式的替代公式。公式(7.8.13)右边应用了熟知标度定律式(7.8.9)两次，一次对 $a$，一次对 $b$。如果估值 $\langle f\rangle_a$ 与 $\langle f\rangle_b$ 是用简单蒙特卡罗均匀随机取样点计算的话，这将是对的。然而，事实上这两个平均估计值是用递归法得到的，所以说明式(7.8.9)成立是没有理由的。不如用以下式代替式(7.8.13)，

$$\mathrm{Var}(\langle f\rangle') = \frac{1}{4}\left[\frac{\mathrm{Var}_a(f)}{N_a^\alpha} + \frac{\mathrm{Var}_b(f)}{(N-N_a)^\alpha}\right] \tag{7.8.22}$$

这里 $\alpha$ 是一个未知常数 $\geqslant 1$（$=1$ 对应简单蒙特卡罗）。在这种情况下，经简单推导可以证明当

$$\frac{N_a}{N} = \frac{\mathrm{Var}_a(f)^{1/(1+\alpha)}}{\mathrm{Var}_a(f)^{1/(1+\alpha)} + \mathrm{Var}_b(f)^{1/(1+\alpha)}} \tag{7.8.23}$$

时，$\mathrm{Var}(\langle f\rangle')$ 有极小值，其极小值为

$$\mathrm{Var}(\langle f\rangle') \propto \left[\mathrm{Var}_a(f)^{1/(1+\alpha)} + \mathrm{Var}_b(f)^{1/(1+\alpha)}\right]^{1+\alpha} \tag{7.8.24}$$

当 $\alpha=1$ 时，方程(7.8.22)至(7.8.24)还原成(7.8.13)至(7.8.15)。为找出 $\alpha$ 的自相容值，用数值实验发现 $\alpha=2$。也就是当 $\alpha=2$ 作为限定取样机会时，应用方程式(7.8.23)递归，观察到的 RSS 算法方差近似按 $N^{-2}$ 变化；而以 $\alpha$ 的其他值代入式(7.8.23)时，情况会更恶劣。（然而，对 $\alpha$ 的敏感性并不算很高；但也不知 $\alpha=2$ 究竟是严格分析的结果，还是仅有的一种有用的直观推断。）

到现在为止，**miser** 的实现与上面描述的算法的主要区别在于式(7.8.23)右边的方差如何估值，我们从实验中发现，以取样函数极大与极小值之差的平方代替纯粹的取样二阶矩是更有益的。当然，估计值随着取样量的增加也增加；而式(7.8.23)用它仅仅是比较近似相等取样数的两个子体积($a$ 和 $b$)。当在被积函数作用区中，起始取样仅给出一个点或少数几个点时，"极大减极小"估计值证明了它的价值。很多现实情况，取样点处在更重要的作用区附近，这使"极大减极小"提供的较大取样加权更有益。

包含在这程序中的第二项改进是引入"颤动参数"dith，它的非零值使子体积不在中间精确划分，而是在小数 0.5±dith 处划分，其中符号±是指引入随机数程序内的随机选择。通常可令 dith 为零。然而，如果被积函数有某些特殊的对称性，使作用区正好处在区域的中点，或区域二次幂因数的中心，则使 dith 不为

零将大有好处。要避免这样的极端情形，作用区被划分为 $d$ 维空间紧密相邻的 $2^d$ 个小块。在这种情况 dith 非零的典型值取 0.1 是有益的。当然，在颤动参数为非零时，必须考虑了体积的不同大小；程序通过变量 fracl 来做这件事情。

该程序还有最后一个特性要阐明，RSS 算法使用单组取样点在所有 $d$ 个方向计算式(7.8.23)，在递归底层，取样点数可能非常少。虽然稀少，但有时也会出现在某一方向的所有取样点只在体积的某一半中；在这种情况下，此方向作为分叉宁愿被忽略，更少见的情况是，在所有方向上，所有取样点都只处在总体积的一半中。在这种情况下，可选择一个随机的方向。如果这种情况在工作中经常发生，就应该增加 MNPT(见程序中的 if(!jb)…)。

正如已给出的 **miser** 返回 ave 作为平均函数值《$f$》的一个估计值，不是 $f$ 在整个范围内的积分，选用其他约定，程序 **vegas** 返回 tgral 作为这个积分。由于长方体体积 V 是已知的，这两个约定自然用(7.8.8)式表示成一般关系式。

```
#include <stdlib.h>
#include <math.h>
#include "nrutil.h"
#define PFAC 0.1
#define MNPT 15
#define MNBS 60
#define TINY 1.0e-30
#define BIG 1.0e30
```

这里 PFAC 是剩余函数计算的一部分，用在每一步来确定函数的方差，至少 MNPT 函数的计算在任何子区域末端完成；仅当 MNBS 函数的计算是可行的，子区域才将进一步一分为二，我们取 MNBS=4 * MNPT

```
static long iran=0;

void miser(float (*func)(float []), float regn[], int ndim, unsigned long npts, float dith ,
    float *ave, float *var)
```

此程序实现在以 regn[1…2 * ndim]指定的长方体内，对用户提供的 ndim 维函数 func 进行蒙特卡罗取样；这个 regn[1...2 * ndim]向量由 ndim"的较低-左边"坐标，其后跟随 ndim 的"较高 右边"坐标而组成。这函数全部取样 npts 次，其位置由递归分层取样确定，在这范围内函数的均值是以 ave 返回；ave 的统计不确定性的估计值(标准偏离的平方)以 var 返回。输入参量 dith 按常规设置为零，但若函数的作用区域落入以2的幂次方划分此区域的边界，则应将 dith 设置为 0.1。

```
{
    void ranpt(float pt[], float regn[], int n);
    float *regn_temp;
    unsigned long n,npre,nptl,nptr;
    int j,jb;
    float avel,varl;
    float fracl,fval;
    float rgl,rgm,rgr,s,sigl,siglb,sigr,sigrb;
    float sum,sumb,summ,summ2;
    float *fmaxl,*fmaxr,*fminl,*fminr;
    float *pt,*rmid;

    pt=vector(1,ndim);
    if (npts < MNBS) {                          太少点进行二分；直接进行蒙特卡罗
        summ=summ2=0.0;
        for (n=1;n<=npts;n++) {
            ranpt(pt,regn,ndim);
            fval=(*func)(pt);
            summ += fval;
            summ2 += fval * fval;
        }
        *ave=summ/npts;
        *var=FMAX(TINY,(summ2-summ*summ/npts)/(npts*npts));
    }
    else {                                      进行原始(均匀)取样
```

```
    rmid=vector(1,ndim);
    npre=LMAX((unsigned long)(npts*PFAC),MNPT);
    fmaxl=vector(1,ndim);
    fmaxr=vector(1,ndim);
    fminl=vector(1,ndim);
    fminr=vector(1,ndim);
    for (j=1;j<=ndim;j++) {                    对每维初始化左、右边界
        iran=(iran*2661+36979) % 175000;
        s=SIGN(dith,(float)(iran-87500));
        rmid[j]=(0.5+s)*regn[j]+(0.5-s)*regn[ndim+j];
        fminl[j]=fminr[j]=BIG;
        fmaxl[j]=fmaxr[j] = -BIG;
    }
    for (n=1;n<=npre;n++) {                    在取样点上循环
        ranpt(pt,regn,ndim);
        fval=(*func)(pt);
        for (j=1;j<=ndim;j++) {                对每维找出左、右边界
            if (pt[j]<=rmid[j]) {
                fminl[j]=FMIN(fminl[j],fval);
                fmaxl[j]=FMAX(fmaxl[j],fval);
            }
            else {
                fminr[j]=FMIN(fminr[j],fval);
                fmaxr[j]=FMAX(fmaxr[j],fval);
            }
        }
    }
    sumb=BIG;                                  选取哪维jb去二分
    jb=0;
    siglb=sigrb=1.0;
    for (j=1;j<=ndim;j++) {
        if (fmaxl[j] > fminl[j] && fmaxr[j] > fminr[j]) {
            sigl=FMAX(TINY,pow(fmaxl[j]-fminl[j],2.0/3.0));
            sigr=FMAX(TINY,pow(fmaxr[j]-fminr[j],2.0/3.0));
            sum=sigl+sigr;                     方程式(7.8.24)，见正文
            if (sum<=sumb) {
                sumb=sum;
                jb=j;
                siglb=sigl;
                sigrb=sigr;
            }
        }
    }
    free_vector(fminr,1,n);
    free_vector(fminl,1,n);
    free_vector(fmaxr,1,n);
    free_vector(fmaxl,1,n);
    if (!jb) jb=1+(ndim*iran)/175000;          MNPT 可以很小
    rgl=regn[jb];                              在左和右之间按比例分配其余取样点
    rgm=rmid[jb];
    rgr=regn[ndim+jb];
    fracl=fabs((rgm-rgl)/(rgr-rgl));
    nptl=(unsigned long)(MNPT+(npts-npre-2*MNPT)*fracl*siglb
        /(fracl*siglb+(1.0-fracl)*sigrb));                 方程式(7.8.23)
    nptr=npts-npre-nptl;
    regn_temp=vector(1,2*ndim);                现在分配和积分这两个子区域
    for (j=1;j<=ndim;j++) {
        regn_temp[j]=regn[j];
        regn_temp[ndim+j]=regn[ndim+j];
    }
    regn_temp[ndim+jb]=rmid[jb];
    miser(func,regn_temp,ndim,nptl,dith,&avel,&varl);
    regn_temp[jb]=rmid[jb];                    递归调用：这里将最终返回
    regn_temp[ndim+jb]=regn[ndim+jb];
    miser(func,regn_temp,ndim,nptr,dith,ave,var);
    free_vector(regn_temp,1,2*ndim);
    *ave=fracl*avel+(1-fracl)*(*ave);
```

```
    *var=fracl*fracl*varl+(1-fracl)*(1-fracl)*(*var);
    Combine left and right regions by equation (7.8.11) (1st line).
    free_vector(rmid,1,n);
  }
  free_vector(pt,1,n);
}
```

miser 程序调用一个短函数 ranpt，用在一特定的 $d$ 维区域内以得到随机点。下述的改进程序 **ranpt**，采用连续调用得到一致随机数生成程序，并做出明确的注释。在这里，通过准随机程序 **sobseq**（第7.7节），可以很容易地修改 **ranpt** 产生随机点，其实，具有 **sobseq** 的 **miser** 程序比具有一致随机偏差的 **miser** 更精确。然而，由于 RSS 的使用与准随机数的使用是完全独立的，所以我们没有使在这里给出的程序依赖 **sobseq**。对重要取样也可给出类似的说明。它在原则上能与 RSS 混合使用。（尽管程序可能极为复杂，但原则上重要取样也可与 **vegas** 及 **miser** 混合使用。）

```
extern long idum;

void ranpt(float pt[], float regn [], int n)
    在 n 维长方形范围中返回均匀随机点 pt。它用于 miser；为了一致偏离调用程序 ran1，用户的主程序中应该初始化全
    程变量 idum，使其为负的种子数。
{
    float ran1(long  * idum);
    int j;

    for (j=1;j<n;j++)
        pt[j]=regn[j]+(regn[n+j]-regn[j]*ran1(&idum);
}
```

## 参考文献和进一步读物：


Kalos，M. H. and Whitlock，P. A. 1986，*Monte Carlo Methods*(New York：Wiley).

Lepage，G. P. 1978，*Journal of Computational Physics*，vol. 27，pp. 192～203. [1]

Lepage，G. P. 1980，"VEGAS：An Adaptive Multidimensional Integration Program，" Publication CLNS-80/447，Cornell University. [2]

Press，W. H.，and Farrar，G. R. 1990，*Computers in Physics*，vol. 4，pp. 190～195. [3]

# 第八章 排 序

## 8.0 引言

本章的内容几乎都没有归入一本数值算法的书中。然而，对于任何优秀的程序员来说，排序技术是应具有的一部份必不可少的实际知识。因此，我们不希望那些已自认为是精通数值计算技术的专家，却忽略了排序这样一类基础课题。

在人们处理数据(实验数据或数值生成的数据)时，常常需要进行排序。人们需要有数据表格或列表，来表示一个或多个独立(或者"控制")的变量，以及一个或多个应变量(或者"所测量"的量)。在各种不同情况下，人们可能希望按这些变量中一个或另一个的顺序来排列这些数据。或者，有时人们可能只希望知道这些数值表中的"中间值"或"四分位"值，后者的任务称为**挑选**，它是与排序紧密相关的。

更明确地说，本章将处理以下一些内容：

· 排序，即按数值的顺序重排数组。

· 按数值顺序重排一个数组的同时，对另外一个或几个数组执行相应的重排，而使所有数组中元素之间的对应关系保持不变。

· 给定一数组，准备一个**索引表**，即指针表，它指明哪一个数组元素按数序排第一、哪一个排第二，等等。

· 给定一数组，准备一个**秩表**，它能指明第一个数组元素的数值秩是什么，第二个数组元素的数值秩是什么，等等。

· 从数组中选取第 $M$ 大的元素。

对于 $N$ 个元素排序的基本任务来说，最佳算法所需的运算量为 $N\log_2 N$ 的几倍。算法发明者试图将这一估计或前面的常数变得尽可能地小。两个最佳算法是由 C. A. R. Hoare 发明的、无可伦比的**快速排序法**(第8.2节)，以及由 *J. W. J* 发明的**堆积排序法**(第8.3节)。

对于大数 $N$(如 $N>1000$)，在大多数机器上，快速排序法要快 1.5 或 2 倍；然而，它需要增加一点额外的存储量，而且是一个较复杂的程序。堆积排序法是一个真正的"同址排序"，因而它编程更加紧凑，更容易略加修改而适应特殊的目的。均衡考虑，我们推荐快速排序法，但我们仍将给出这两算法的实施程序。

对于小数 $N$，如果前面的常数足够小的话，最好选用运算量是 N 的较高也较劣的阶幂次的算法。对于 $N<20$，**直接插入法**(第8.1节)是简捷和足够快的。我们将它包括进来有一些担心：它是一个 $N^2$ 的算法，其潜在的误用可能性(将它用于太大的 $N$)很大。它所耗用的机时是如此可观，以至于我们不打算将任何 $N^2$ 算法包括进来。然而，我们将把界线划在**气泡排序法**这一低效的 $N^2$ 算法处，虽它是基本计算机科学书中所喜爱的方法。如果读者知道什么是气泡排序法，就将它从头脑中抹去；如果还不知道，就请记住永远别去用它！

对于 $N>50$，**剥壳法**(第8.1节)只比直接插入法稍复杂，与更复杂的快速排序法相比更

有竞争性，这种方法在最坏情况下是 $N^{3/2}$ 运算量，但通常更快。

想要了解排序领域的进一步的知识和详细内容，参阅[1,2]。

参考文献和进一步读物：


Knuth, D. E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley). [1]

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley), Chapters 8~13. [2]


## 8.1 直接插入法和剥壳法

**直接插入法**是一种 $N^2$ 运算量的程序，因而只能用在 $N$ 较小时，比如 $N<20$。

这种技巧完全是，有经验的玩扑克者洗牌时对牌排序所用的方法：挑出第二张牌将它按与第一张的顺序插入；然后挑出第三张将它按大小顺序插到前两张中；如此下去直到将最后一张拿起并插入。

```
vioid piksrt(int n, float arr[])
    用直接插入法将数组 arr[1..n]按递增顺序排列。n 是输入量；输出时 arr 被排序好的序列替换。
{
    int i,j;
    float a;

    for (j=2;j<=n;j++) {                         依次挑出每个元素
        a=arr[j];
        i=j-1;
        while (i > 0  &&  arr[i]  >  a){          寻找它的插入位置
            arr[i+1]=arr[i];
            i--;
        }
        arr[i+1]=a;                               将其插入
    }
}
```

如果想在排序 **arr** 的同时重整一个数组 **brr**，就可简单地在移动 **arr** 的一个元素的同时，移动一个 **brr** 的元素。

```
void piksr2(int n, float arr[], float brr[])
    用直接插入法将序列 arr[1..n]按递增顺序排列，同时对序列的 brr[1..n]做相应的调整。
{
    int i,j;
    float a,b;

    for (j=2;j<=n;j++){                           依次挑出每个元素
        a=arr[j];
        b=brr[j];
        i=j-1;
        while (i > 0  &&  arr[i] > a)  {          寻找插入的位置
            arr[i+1]=arr[i];
            brr[i+1]=brr[i];
            i--;
        }
        arr[i+1]=a;                               将它插入
        brr[i+1]=b;
```

    }
}

对于在排序一个数组的同时有很多数组需要重排的情况，参见第8.4节。

### 8.1.1 剥壳法

这实际上是直接插入法的一个变种，但确实是一种非常强有力的变种。大致的思想是，例如对排序 16 个数 $n_1 \ldots n_{16}$的情况，是如下进行的：第一次排序，用直接插入法依次排序 8 组数$(n_1,n_9)$，$(n_2,n_{10})$，…，$(n_8,n_{16})$。然后依此排序 4 组数$(n_1,n_5,n_9,n_{13})$，…，$(n_4,n_8,n_{12},n_{16})$。之后依次排序两个各有 8 个数的数组，前一个是$(n_1,n_3,n_5,n_7,n_9,n_{11},n_{13},n_{15})$。最后，将全部 16 个数排序。

当然，只有最后一次排序是将数按顺序排列所必需的。那么，前面的部分排序是什么目的呢？答案是，前面的排序使数字有效地上下移动，以接近它们的最终位置。因而，最后一次排序实施直接插入法时，实际只需移动少数几个元素。(设想对一手已经基本上按顺序排列的牌进行排序。)

每次经过对数据排序后数字之间的间隔(上例中为 8,4,2,1)称之为增量，所以，有时剥壳法又称为减小增量排序法。对于如何选择一组好的增量已有了大量的研究，但最佳选择还未找到。实际上，… 8,4,2,1 这组并非是好的选择，特别是对于 $N$ 是 2 的幂次时。一个较好的选择是序列

$$(3^k - 1)/2, \ldots, 40, 13, 4, 1 \tag{8.1.1}$$

它由递推产生

$$i_1 = 1, \qquad i_{k+1} = 3i_k + 1, \qquad k = 1,2,\ldots \tag{8.1.2}$$

可以证明(参阅[1])，采用这一序列的增量，对于最坏可能顺序的原始数据，上述过程所需的总运算量是 $N^{3/2}$量级。因此，对于“随机”顺序的数据，至少对于 $N<60000$，其运算量近似为 $N^{1.25}$。对于 $N>50$，不管怎样，一般快速排序法是更快的。程序如下：

```
void shell(unsigned long n, float a[])
    用剥壳算法(减小增量排序)将数组 a[1..n]按递增顺序排列，n 是输入量；输出时 a 被排序好的数组替换。
{
    unsigned long i,j,inc;
    float v;
    inc=1;                              确定初始增量
    do {
        inc *= 3;
        inc++;
    } while (inc <= n);
    do {                                部份排序循环
        inc /= 3;
        for (i=inc+1;i<=n;i++) {        直接插入的外循环
            v=a[i];
            j=i;
            while (a[j-inc] > v) {      直接插入的内循环
                a[j]=a[j-inc];
                j -= inc;
                if (j <= inc) break;
            }
            a[j]=v;
```

```
    }
  } while (inc > 1);
}
```

参考文献和进一步读物：

Knuth, D. E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programmin*(Reading. MA: Addison-Wesley), §5.2.1. [1]

## 8.2 快速排序法

在大多数机器上，平均说来对于大 $N$，快速排序法是一种最快的排序算法。这是一种"划分—交换"的排序方法：一个划分元素 a 被从序列中挑选出来。然后经过成对地交换元素，原先的序列被划分成两个子数组。在一次划分结束之后，元素 a 已经处于它在数组中的最后位置。在它左面的子数组的元素全都 $\leqslant$ a，而右边子数组的元素全都 $\geqslant$ a。进而，在左右子序列中分别继续进行这种划分，如此下去。

划分的过程首先得选出一个元素，比如说最左边的元素，将其当作划分元素 a。用一个指针从上向下扫过数组直到发现一个 $>$ a 的元素，然后用另一个指针从数组尾部向上扫过数组直到发现一个 $<$ a 的元素。这两个元素显然不在最后划分完之后它们应该处的位置上，于是交换它们。继续这一过程直到两指针交汇。交汇处便是应插入 a 的位置，这一轮划分便结束了。当一个元素与划分元素相等时应该如何做？这个问题是很微妙的；我们建议参阅 Sedgenick[1] 中的讨论。(答案：应该停下并做一次交换)。

为运行速度，我们在实施快速排序法时不使用递归。因而此算法需要有辅助的存储数组，其长度为 $2\log_2 N$，它被用来当作一个下压堆线来保存那些暂时搁置的子数组。当一个子数组已经被缩减到只有 $M$ 个元素时，用直接插入法会变得更快些(第8.1节)，因而我们也是这第做的。$M$ 值的最佳设置与机器有关，但 $M=7$ 总不会太差。有些人喜欢将短的子数组暂不排序留到最后，在最后做一次大的插入排序，因为每个元素最多移动 7 次，这与立刻做排序的效率一样，并且节约总开销。但是，在采用页面存储的现代机器上，一次整个处理一个大数组会增大开销。我们也没有发现将插入排序放到最后做有什么任何好处。

如前所述，快速排序法的平均运行时间是很快的；但它的最坏情况的运行时间可以很慢：对于最坏情况实际上它是一个 $N^2$ 算法！对于最直接地应用快速排序法来说，最坏情况实际上就是输入数组已经基本有序的情况！这种输入数组的有序性很容易在实际情况中出现。避免这种情况的一种方法是，采用一个小的随机数产生器来挑选一个随机元素作为划分元素。另一种方法是，采用当前子数组的第一，中间和最后元素的中间值来替换划分元素。

快速排序法的巨大速度来源于内循环的简单和有效。仅只加一上项不必要的检测(例如，一项保证指针不移出数组的检测)就会使运行时间几乎加倍！人们为了避免这种不必要的检测，在子数组的两端设置"哨兵"。最左边的哨兵 $\leqslant$ a，最右边的 $\geqslant$ a。若采用"三数中值法"来挑选划分元素，则我们可以运用那两个不是中值的元素做为这一子序列的哨兵。

下面是我们的实施程序[1]：

```
#include "nrutil.h"
#define SWAP(a,b) temp=(a);(a)=(b);(b)=temp;
#define M 7                              是使用直接插入法排序的子数组的大小
#define NSTACK 50                        是所需的辅助存储器
```

```
void sort(unsigned long n, float arr[])
    用快速排序法使数组 arr[1..n]按递增排列,n 是输入值,arr 在输出时由已排序的数组替换
{
    unsigned long i,ir=n,j,k,l=1;
    int jstack=0,*istack;
    float a,temp;

    istack=ivector(1,NSTACK);
    for (;;) {                                  当子数组足够小时插入排序
        if (ir-l < M) {
            for (j=l+1;j<=ir;j++) {
                a=arr[j];
                for (i=j-1;i>=1;i--) {
                    if (arr[i] <= a) break;
                    arr[i+1]=arr[i];
                }
                arr[i+1]=a;
            }
            if (jstack == 0) break;
            ir=istack[jstack--];                弹出堆栈开始新一轮划分
            l=istack[jstack--];
        } else {
            k=(l+ir) >> 1;                      挑选左、中、右三元素的中值作为划分元素 a 同
            SWAP(arr[k],arr[l+1])                   时重新排列以使 a[l+1] ≤ a[l] ≤ a[ir]
            if (arr[l+1] > arr[ir]) {
                SWAP(arr[l+1],arr[ir])
            }
            if (arr[l] > arr[ir]) {
                SWAP(arr[l],arr[ir])
            }
            if (arr[l+1] > arr[l]) {
                SWAP(arr[l+1],arr[l])
            }
            i=l+1;                              为划分初始化指针
            j=ir;
            a=arr[l];                           划分元素
            for (;;) {                          开始内循环
                do i++; while (arr[i] < a);         向下扫描寻找元素>a
                do j--; while (arr[j] > a);         从尾向上扫描寻找元素<a
                if (j < i) break;               指针交汇，划分结束
                SWAP(arr[i],arr[j]);            交换元素
            }                                   结束内循环
            arr[l]=arr[j];                      插入划分元素
            arr[j]=a;
            jstack += 2;
            将指针压入堆栈中更大的子数组，立刻处理较小的子数组
            if (jstack > NSTACK) nrerror("NSTACK too small in sort.");
            if (ir-i+1 >= j-l) {
                istack[jstack]=ir;
                istack[jstack-1]=i;
                ir=j-1;
            } else {
                istack[jstack]=j-1;
                istack[jstack-1]=l;
                l=i;
            }
        }
    }
    free_ivector(istack,1,NSTACK);
}
```

可以在排序 arr 的同时，重排其他任何数组。程序基本是前一程序的重复。

```
#include "nrutil.h"
```

```
#define SWAP(a,b) temp=(a);(a)=(b);(b)=temp;
#define M 7
#define NSTACK 50

void sort2(unsigned long n, float arr[], float brr[])
用快速排序法使数组arr[1..n]按递增排列,而同时相应地重排数组brr[1..n]。
{
    unsigned long i,ir=n,j,k,l=1;
    int *istack,jstack=0;
    float a,b,temp;
    istack=ivector(1,NSTACK);
    for (;;) {                              当子数组足够小时用插入排序
        if (ir-l < M) {
            for (j=l+1;j<=ir;j++) {
                a=arr[j];
                b=brr[j];
                for (i=j-1;i>=1;i--) {
                    if (arr[i] <= a) break;
                    arr[i+1]=arr[i];
                    brr[i+1]=brr[i];
                }
                arr[i+1]=a;
                brr[i+1]=b;
            }
            if (!jstack) {
                free_ivector(istack,1,NSTACK);
                return;
            }
            ir=istack[jstack];              弹出堆栈开始新一轮划分
            l=istack[jstack-1];
            jstack -= 2;
        } else {
            k=(l+ir) >> 1;                  挑选左、中、右三元素的中值作为划分元素a同
            SWAP(arr[k],arr[l+1])               时重新排列以使a[l+1] ≤ a[l] ≤ a[ir]
            SWAP(brr[k],brr[l+1])
            if (arr[l+1] > arr[ir]) {
                SWAP(arr[l+1],arr[ir])
                SWAP(brr[l+1],brr[ir])
            }
            if (arr[l] > arr[ir]) {
                SWAP(arr[l],arr[ir])
                SWAP(brr[l],brr[ir])
            }
            if (arr[l+1] > arr[l]) {
                SWAP(arr[l+1],arr[l])
                SWAP(brr[l+1],brr[l])
            }
            i=l+1;                          为划分初始化指针
            j=ir;
            a=arr[l];                       划分元素
            b=brr[l];
            for (;;) {                      开始内循环
                do i++; while (arr[i] < a);        从数组首向下扫描寻找元素>a
                do j--; while (arr[j] > a);        从数组尾向上扫描寻找元素<a
                if (j < i) break;           指针交汇,划分结束
                SWAP(arr[i],arr[j])         交换两个数组的元素
                SWAP(brr[i],brr[j])
            }                               结束内循环
            arr[l]=arr[j];                  在两个数组中插入划分元素
            arr[j]=a;
            brr[l]=brr[j];
            brr[j]=b;
            jstack += 2;
            将指针压入堆栈中更大的子数组,立刻处理较小的子数组
            if (jstack > NSTACK) nrerror("NSTACK too small in sort2.");
            if (ir-i+1 >= j-l) {
```

```
            istack[jstack]=ir;
            istack[jstack-1]=i;
            ir=j-1;
        } else {
            istack[jstack]=j-1;
            istack[jstack-1]=l;
            l=i;
        }
      }
    }
}
```

从原则上说，可以同 brr 一起重排任何数目的附加数组，但在数组的数目增多时，这样做是很浪费的。有效的办法是，使用第8.4节中所讲的索引表的方法。

参考文献和进一步读物：


Sedgwick, R. 1978, *Communications of the ACM*, vol. 21, pp. 847~857. [1]


## 8.3 堆积排序法

通常，堆积排序法不如快速排序法快，但它是我们喜爱的一种排序法。它是真正的"同址"排序，不要求附加的存储单元。它是一个 $N\log_2 N$ 次的过程，这不仅从平均上来说，而且对于输入数据是最坏情况的次序，它也是如此。事实上，最坏情况下只比平均运行时间多百分之二十左右。

对于堆积排序法给出完整的理论说明已超出了本书的范围。现在，我们只将提及它的基本原理，如果读者想了解细节可参阅文献[1,2]，或者自己分析程序。

一系列 $N$ 个数 $a_i$, $i=1, \ldots, N$，如果满足下列关系就说它形成一个"堆"

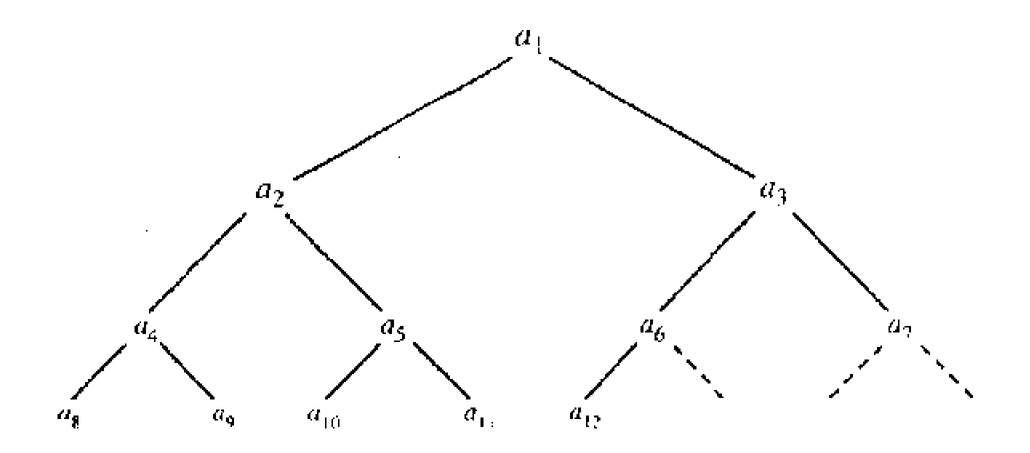
$$a_{j/2} \geq a_j \quad \text{对于} \quad 1 \leq j/2 < j \leq N \tag{8.3.1}$$

这里的除法 $j/2$是指"整数除法"，也就是说是一个确切的整数或是略去小数位的最近整数。定义(8.3.1)是有意义的，如果将数字 $a_i$ 看成是按一个二元树排列的，在最上面的"老板"结点是 $a_1$，两个"下属"结点是 $a_2$和 $a_3$，它们的四个下属结点是 $a_4$到 $a_7$，等等。(参见图8.3.1)按这种形式，这一等级制度从下而下，每个"管理者"大于或等于它的两个"被管理者"。

如果已经将数组设法重整成堆的形式，那么将它排序就很简单了：将"堆顶"除去，即使不排序也知道它是最大的。然后将它的最大的下属元素"提升"到堆顶。然后再提升它的最大下属元素，如此继续。这一过程很像在一个大公司中总裁退休时会发生(或假设会发生)的情况。然后可以使新的总裁退休而重复上述过程。最终整个事件是一个 $N\log_2 N$ 过程，因为每个总裁退休将导至 $\log_2 N$ 次下属的提升。

但开始时如何将数组整理成一个堆呢？答案仍然是一个很像公司中提升的"上移"过程。假想公司开始时有 $N/2$个在生产线上的雇员，但没有管理人员。于是一个管理人员被雇来管理两个工人。如果他没有那两个工人中的任何一个有能力，则工人中的一个被提升为管理员，而原管理员降级到生产线上。管理员雇完之后，就开始雇管理人员的管理者，以此类推，直到公司的阶梯的最高层。每个雇员从这棵树的顶部被带进来，然后立刻向下筛选，更有能

力的工人得到提升直到停在他合适的阶层上。




由向上的通路所连接的元素是相对排序的，但“侧面”相邻的元素却没有必然的大小关系。

图8.3.1　12个元素的“堆”所隐含的顺序

在应用堆积排序法时，同样的“上移”代码段可以应用在开始产生堆时，也可以用在随后的退休和提升过程中。堆积排序法的一个完整实现，代表了一个大公司的整个生存循环：$N/2$ 个工人被雇来；$N/2$ 个潜在的管理者被雇来；有一个职位的升迁过程，它是一种超级 Peter 原理的排序；最后，按注定的过程，每个原来的雇员被依次提升为总裁。

```
void hpsort(unsigned long n, float ra[])
    用堆积排序法将数组ra[1..n]按递增顺序排列。n 是输入量；输出时 ra 被排序好的数组替换。
{
    unsigned long i,ir,j,l;
    float rra;

    if (n < 2) return;
    l=(n >> 1)+1;         指标l将在“雇佣”(堆积形成)过程中从它的初始值递减直到减为1。当它减到1后，
    ir=n;                 指标ir将在“退休和提升”(堆分解)过程中从它的初始值递减直到1。
    for (;;) {
        if (l > 1)                                  仍在雇佣过程中
            rra=ra[--l];
        }else{                                      在退休和提升过程中
            rra=ra[ir];                             在数组尾部清出一个空间
            ra[ir]=ra[1];                           使堆顶退休并存入其中
            if (--ir == 1) {                        最后一次提升完成
                ra[1]=rra;                          所有工作者中最无竞争能力的一个！
                break;
            }
        }
        i=l;                                        不管是在雇佣过程中还是在提升过程中，在此我们
        j=l+l;                                      设置并移动元素 rra 到它的合适阶层上
        while (j <= ir) {
            if (j < ir && ra[j] < ra[j+1]) j++;     与下属中较大的比较
                if (rra < ra[j]) {                  将 rra 降级
                    ra[i]=ra[j];
                    i=j;
                    j <<= 1;
                }else j=ir+1;                       这是 rra 的层次，设置 j 来终止下移
        }
        ra[i]=rra;                                  将 rra 归位
```

```
    }
}
```

## 8.4 索引和分秩

在数据文件管理中，**关键**这个概念起着重要的作用。在数据文件中一条数据记录可包含若干项，或称**域**。例如，天气观察文件中的一条记录可以有记录的时间、温度、和风速的域。当排序记录时，必须决定哪个域要顺序排列。记录中其它域只是跟着移动，这些域通常在最后不形成某种特殊的顺序。这个执行排序的域称之为**关键域**。

对于一个有很多记录和很多域的数据文件，要将 N 条记录按照它们的关键数组 $K_i$，$i=1,\ldots,N$ 排列成序，其工作量很大，实现很困难。但是，可以代之用构造**索引表** $I_j$，$j=1,\ldots,N$，使得最小的 $K_i$ 有 $i=I_1$，第二小的有 $i=I_2$，以此类推，直至对于最大的 $K_i$ 有 $i=I_N$。换言之，数组

$$K_{I_j} \qquad j=1,2,\ldots,N \tag{8.4.1}$$

是以 $j$ 为索引的顺序排列。当有一索引表后，就不需要将记录从它原来的顺序中移动。进一步地，同样的一组记录可以产生不同的索引表，这是根据不同的关键域产生的。

产生一个索引表的算法是很直接的：将索引数组初始化成整数1到 $N$，之后用快速排序法，像排序关键数组一样地移动其元素。于是关键数组中最小元素所对应的序号最终停在第一位，依此等等。

```
#include "nrutil.h"
#define SWAP(a,b) temp=(a);(a)=(b);(b)=temp;
#define M 7
#define NSTACK 50

void indexx(unsigned long n, float arr[], unsigned long indx[])
    索引一个数组 arr[1..n]，也就是输出一个数组 indx[1..n]使得 arr[indx[j]]对于 j=1,2,...,N 是递增的。输入量
    n 和 arr 不改变。
{
    unsigned long i,indxt,ir=n,itemp,j,k,l=1;
    int jstack=0,*istack;
    float a;
    istack=ivector(1,NSTACK);
    for (j=1;j<=n;j++) indx[j]=j;
    for (;;) {
        if (ir-l < M) {
            for (j=l+1;j<=ir;j++) {
                indxt=indx[j];
                a=arr[indxt];
                for (i=j-1;i>=1;i--) {
                    if (arr[indx[i]] <= a) break;
                    indx[i+1]=indx[i];
                }
                indx[i+1]=indxt;
            }
            if (jstack == 0) break;
            ir=istack[jstack--];
            l=istack[jstack--];
        } else {
            k=(l+ir) >> 1;
            SWAP(indx[k],indx[l+1]);
```

```
            if (arr[indx[l+1]] > arr[indx[ir]]) {
                SWAP(indx[l+1],indx[ir])
            }
            if (arr[indx[l]] > arr[indx[ir]]) {
                SWAP(indx[l],indx[ir])
            }
            if (arr[indx[l+1]] > arr[indx[l]]) {
                SWAP(indx[l+1],indx[l])
            }
            i=l+1;
            j=ir;
            indxt=indx[l];
            a=arr[indxt];
            for (;;) {
                do i++; while (arr[indx[i]] < a);
                do j--; while (arr[indx[j]] > a);
                if (j < i) break;
                SWAP(indx[i],indx[j])
            }
            indx[l]=indx[j];
            indx[j]=indxt;
            jstack += 2;
            if (jstack > NSTACK) nrerror("NSTACK too small in indexx.");
            if (ir-i+1 >= j-l) {
                istack[jstack]=ir;
                istack[jstack-1]=i;
                ir=j-1;
            } else {
                istack[jstack]=j-1;
                istack[jstack-1]=l;
                l=i;
            }
        }
    }
    free_ivector(istack,1,NSTACK);
}
```

如果想要在排序一个序列的同时，将几个或很多其它数组做相应的调整，则应该首先产生一个索引表，然后利用它依次重整每个数组。这就需要两个工作空间：一个放置索引，一个用来做暂时的移动，然后将排好顺序的数组存回原数组中。对于三个数组的情况，程序如下：

```
#include "nrutil.h"

void sort3(unsigned long n, float ra[], float rb[], float rc[])
    将数组 ra[1..n]排序成递增顺序，同时将数组 rb[1..n]和 rc[1..n]做相应调整。通过程序 indexx 构成一个索引表。
{
    void indexx(unsigned long n, float arr[], unsigned long indx[]);
    unsigned long j, *iwksp;
    float *wksp;

    iwksp=lvector(1,n);
    wksp=vector(1,n);
    indexx(n,ra,iwksp);                                  制作引表
    for (j=1;j<=n;j++)  wksp[j]=ra[j];                   存储数组 ra
    for (j=1;j<=n;j++)  ra[j]=wksp[iwksp[j]];            复制它并重排返回
    for (j=1;j<=n;j++)  wksp[j]=rb[j];                   依此处理 rb
    for (j=1;j<=n;j++)  rb[j]=wksp[iwksp[j]];            依此处理 rc
    for (j=1;j<=n;j++)  wksp[j]=rc[j];
    for (j=1;j<=n;j++)  rc[j]=wksp[iwksp[j]];
    free_vector(wksp,1,n);
```

```
    free_lvector(iwksp,1,n);
}
```

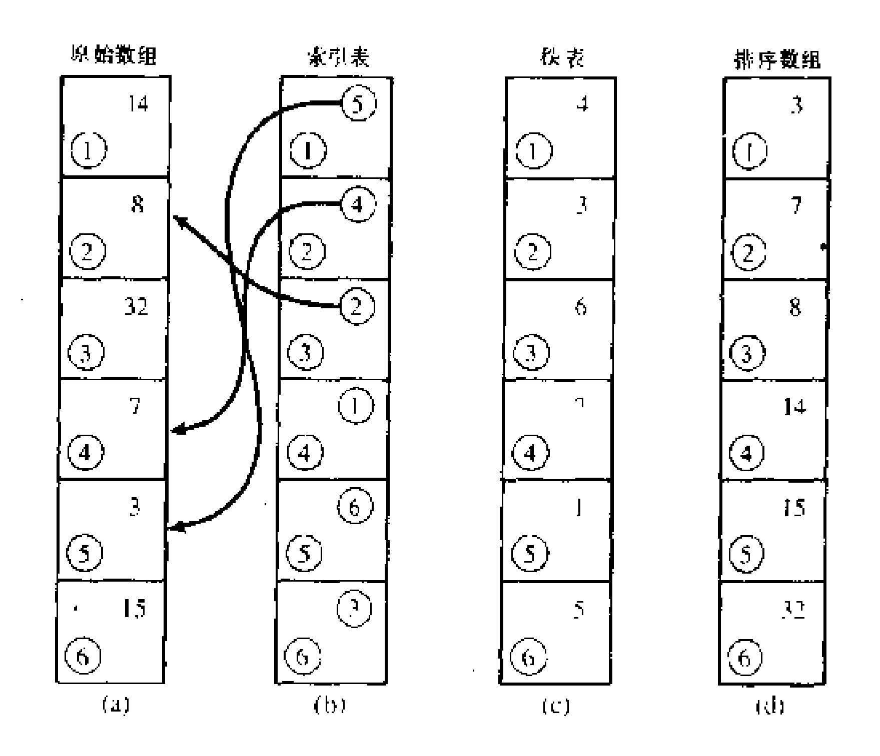
数组数更多的情况显然可直接类推。

一个秩表不同于一个索引表。一个秩表的第 $j$ 个入口给出原先的关键数组中第 $j$ 个元素的秩，其范围是从 1 (如果该元素是最小的)到 $N$(如果该元素最大)。人们能够很容易地从一个索引表产生一个秩表：

```
void rank(unsigned long n, unsigned long indx[], unsigned long irank[])
    给定程序 indexx 输出的索引表数组 indx[1..n]，本程序返回相应的秩表数组 irank[1..n]。
{
    unsigned long j;

    for (j=1;j<=n;j++) irank[indx[j]]=j;
}
```

图8.4.1总结了本节中所讨论的概念。




(a) 一个未排序的六个数的数组。(b) 索引表，其内容是指出(a)中元素按递增顺序排列的指针。(c) 秩表，其内容是对应(a)中元素的秩序号。(d) (a)中元素的排序数组。

图8.4.1

## 8.5 挑选第 M 大的元素

挑选和排序是一对姐妹。排序要求将整个数据数组重排，而挑选则只请求选出一个返回值：在 $N$ 个元素中哪一个是第 $k$ 个小的元素(或等效地，第 $m=N+1-k$ 大的元素)？遗憾的是，为了挑选的计算目的，挑选的最快方法需重整数组，典型的是将所有较小的元素置于

第 $k$ 个之左，所有较大元素置于右，并且在每一个子集内顺序是杂乱的。这种副作用轻者是无害的，重者就十分的不方便。当序列很长时，以至于打印出一份来就很费时，即使挑选所带来的计算上的负担，只是一个更大计算量中可忽略的一部分时，人们还是喜欢选用没有副作用的挑选方法，这种方法不打乱原数组的顺序。这种定点挑选法比更快的方法慢约10倍。下面我们给出两种程序。

挑选通常用于对一组数据进行统计定性。人们通常想知道一列数据的中间值，或上四分之一或下四分之一值。当 $N$ 是奇数时，中间值就是第 $k$ 个元素，其中 $k=(N+1)/2$ 。当 $N$ 是偶数时，统计书上定义中间值是 $k=N/2$ 和 $k=N/2+1$两元素（也就是从上数第 $N/2$ 和从下数第 $N/2$）的算术平均值。如果接受这种规定，就必须做两次分别的挑选，来找到这些值。对于 $N>100$ ，我们通常定义 $k=N/2$ 为中间元素，不按这种规定。

若允许重排数组，最快的挑选方法则就是**划分法**，与快速排序法（第8.2节）中作法完全相同。挑选一个"随机"分割元素，用它来核对整个序列，迫使较小的元素排列左边，较大的排到右边。正如在快速排序法中那样，优化内循环是非常重要的，使用"哨兵"（第8.2节）来减少比较的数量。但在排序中，还需对分割出的两个子数组分别继续进行操作。然而在挑选中，我们可以略去一个子数组，而只注意包含了要找的第k 元素的那一个子集。用划分法来挑选就不需要用于操作期间的堆栈，并且它的运算量是 $N$ 量级而不是 $N\log N$ 量级的（参见[1]）。对照第8.2节中的排序程序 **sort**，下列程序就很容易明白：

```
#define SWAP(a,b) temp=(a);(a)=(b);(b)=temp;

float select(unsigned long k, unsigned long n, float arr[])
    返回数组 arr[1..n]中第 k 小的元素。输入的数组将被重排，所挑选的值返回在 arr[k]中，所有小的元素在 arr[1..k
    -1]中（以任意顺序），而所有大的元素都在 arr[k+1..n]中（以任意顺序）。
{
    unsigned long i,ir,j,l,mid;
    float a,temp;

    l=1;
    ir=n;
    for (;;) {
        if (ir <= l+1) {                    激活包含 1 个或 2 个元素的划分
            if (ir == l+1 && arr[ir] < arr[l]) {     两个元素的情况
                SWAP(arr[l],arr[ir])
            }
            return arr[k];
        } else {
            mid=(l+ir) >> 1;                挑选左、中、右三元素的中间值作为划分元素a。
            SWAP(arr[mid],arr[l+1])             同时重整以使
            if (arr[l+1] > arr[ir]) {           arr[l+1] ≤ arr[l]
                SWAP(arr[l+1],arr[ir])          arr[ir] ≥ arr[l].
            }
            if (arr[l] > arr[ir]) {
                SWAP(arr[l],arr[ir])
            }
            if (arr[l+1] > arr[l]) {
                SWAP(arr[l+1],arr[l])
            }
            i=l+1;                          为划分初始化指针
            j=ir;
            a=arr[l];                       划分元素
            for (;;) {                      开始内循环
                do i++; while (arr[i] < a);     从数组首部向下扫描寻找元素>a
                do j--; while (arr[j] > a);     从数组尾向上扫描寻找元素<a
```

```
            if (j < i) break;                    指针交汇，划分结束
            SWAP(arr[i],arr[j])
        }                                        结束内循环
        arr[l]=arr[j];                           插入划分元素
        arr[j]=a;
        if (j >= k) ir=j-1;                      使包含第 k 元素的划分继续进行
        if (j <= k) l=i;
    }
  }
}
```

定点挑选或称非破坏挑选在概念上很简单，但它需要很多记录，因而相对较慢。基本的想法是，随机地选出 $M$ 个元素，将其排序，然后将整个数组通过一遍，数一数在 $M+1$ 个空隙中分别有多少元素。包含第 $k$ 个大的元素的空隙叫做"活"空隙。然后再进行第二轮，首先在活空隙中随机地挑选 $M$ 个元素，再确定当前活元素落入新的，更细的 $M+1$ 个空隙中的哪一个。以此类推，直到第 $k$ 元素定位在一个大小为 $M$ 的单个数组中，这时直接挑选是可能的了。

我们如何挑选 $M$? 操作的循环次数为 $\log_M N=\log_2 N/\log_2 M$ 随着 M 的增大而减小；但将每一元素定位于 $M+1$ 个子空隙的工作量则增大，例如，对于二分法来说是 $\log_2 M$ 量级的大小。在每一轮中需要查看所有 $N$ 个元素，若仅找那些还活着的元素，那二分法主要由发生在第一轮中的 $N$ 所支配。使 $O(N\log_M N)+O(N\log_2 M)$ 取极小值，便得结果

$$M \sim 2^{\sqrt{\log_2 N}} \tag{8.5.1}$$

对数值的平方根变化是很缓慢的，以使对于机器时间的考虑变得重要了。通常我们选用 $M=64$。

在下列程序 **selip** 中有两个附加的小技巧，(i) 将 $M$ 个随机值的集增扩到 $M+1$，其第 $M+1$ 个元素是它们的算术平均值，(ii) 用一种使靠前的元素更容易被选中的方法，在一次"飞过"数据过程中挑选这 $M$ 个随机值。(这种基本想法是，使 $m>M$ 的元素有 $M/m$ 的机会被选中。它能用归纳法证明，它将会产生所需要的结果)。

```
#include "nrutil.h"
#define M 64
#define BIG 1.0e30
#define FREEALL free_vector(sel,1,M+2);free_lvector(isel,1,M+2);

float selip(unsigned long k, unsigned long n, float arr[])  返回数组 arr[1..n]中第 k 小的元素，输入数组不变
{
    void shell(unsigned long n, float a[]);
    unsigned long i,j,jl,jm,ju,kk,mm,nlo,nxtmm,*isel;
    float ahi,alo,sum,*sel;

    if (k < 1 || k > n || n <= 0) nrerror("bad input to selip");
    isel=lvector(1,M+2);
    sel=vector(1,M+2);
    kk=k;
    ahi=BIG;
    alo = -BIG;
    for (;;) {                                 主反复循环直到找到所需元素
        mm=nlo=0;
        sum=0.0;
        nxtmm=M+1;
        for (i=1;i<=n;i++) {                   将整个数组通过一遍
            if (arr[i] >= alo && arr[i] <= ahi) {
               只考虑括号内的元素
```

```
            mm++;
            if (arr[i] == alo) nlo++;        在括号相等的情况中
            现在用统计方法在范围以内等概率地选m个元素，即使事先不知道它们有多少!

            if (mm <= M) sel[mm]=arr[i];
            else if (mm == nxtmm) {
                nxtmm=mm+mm/M;
                sel[1 + ((i+mm+kk) % M)]=arr[i];   %操作提供某种随机数
            }
            sum += arr[i];
        }
    }
    if (kk <= nlo) {                         所需元素与下界相等，返回它
        FREEALL
        return alo;
    }
    else if (mm <= M) {                      所有范围内的元素被保留，于是用直接
        shell(mm,sel);                           方法返回答案
        ahi = sel[kk];
        FREEALL
        return ahi;
    }
    sel[M+1]=sum/mm;                         用平均值扩充(固定退化)所选集合，并排序
    shell(M+1,sel);
    sel[M+2]=ahi;
    for (j=1;j<=M+2;j++) isel[j]=0;          将计数数组置零
    for (i=1;i<=n;i++) {                     将整个数组再过一遍
        if (arr[i] >= alo && arr[i] <= ahi) {        对于范围内的元素
            jl=0;
            ju=M+2;
            while (ju-jl > 1) {              ...用二分法找寻它的位置
                jm=(ju+jl)/2;                          ...
                if (arr[i] >= sel[jm]) jl=jm;
                else ju=jm;
            }
            isel[ju]++;                      ...并计数器加1
        }
    }
    j=1;                                     现在我们能够将范围缩小一级，即缩小m
    while (kk > isel[j]) {                       级因子
        alo=sel[j];
        kk -= isel[j++];
    }
    ahi=sel[j];
  }
}
```

大概的时间估算：**selip** 大的比 **select** 慢十倍。实际上，当 N 在～$10^5$范围内，selip 要比用 sort 进行一次全排序慢 1.5 倍左右，而 select 比 sort 快 6 倍。用户应在时间、存储和方便性之间仔细权衡。

当然，要寻找数组中最大或最小元素这一类小事，不应该采用上述任一种程序。而读者可以自己简单地用 for 循环编程。当与 $N$ 比较 $k$ 是较适中的情况，还有一种好的编程方法，使附加 $k$ 级存储量的负担不是太重。例如，采用堆积排序法(第8.3节)，将整个 $N$ 长的数组通过一遍，而保存 $m$ 个最大的元素。堆积结构的优点是，每个候选人被入选只需比较 $\log m$ 一次，而不是 $m$ 次。这当 $m>O(\sqrt{N})$时，此方法确定需占用相当的内存，但这并不碍事而且容易编程。下列程序展示了这一思想。

```
void hpsel(unsigned long m, unsigned long n, float arr[], float heap[])
```

在 heap[1..m]中返回数组 arr[1..n]的前 m 个最大的元素，并保证 heap[1]是第 $m$ 大的元素。数组 $arr$ 不变。为保证效应，本程序必须仅当 m≪n 时使用。

```
{
    void sort(unsigned long n, float arr[]);
    void nrerror(char error_text[]);
    unsigned long i,j,k;
    float swap;

    if (m > n/2 || m < 1) nrerror("probable misuse of hpsel");
    for (i=1;i<=m;i++) heap[i]=arr[i];
    sort(m,heap);                      通过淘汰产生初始堆。我们假定m<<n
    for (i=m+1 i<=n;i++) {             对每一所剩元素…
        if (arr[i] > heap[1]) {        置于堆上
            heap[1]=arr[i];
            for (j=1;;) {              下移
                k=j << 1;
                if (k > m) break;
                if (k != m && heap[k] > heap[k+1]) k++;
                if (heap[j] <= heap[k]) break;
                swap=heap[k];
                heap[k]=heap[j];
                heap[j]=swap;
                j=k;
            }
        }
    }
}
```

参考文献和进一步读物：


Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley), pp. 126ff. [1]


## 8.6 等价类的确定

一些排序和查找的技巧涉及到数据结构方面的知识，例如树和链接表等，其详细内容已超出了本书的范围。这些结构及其操作是计算机科学的精彩部份，它与数字分析内容截然不同，关于这一主题的书籍也并不缺乏。

在处理实验数据的过程中，我们发现上述结构中一种确定等价类的特殊操作，它在应用中常常出现，所以....我们将它包括进来。

问题是这样的：有 $N$ 个"元素"(或者"数据点"，或是其它任何东西)，编号为1，...，$N$。给出一些表明元素是否属于"同一性"的相同**等价类**的配对信息，这信息可以是根据任何恰好感兴趣的标准得来的。例如，可能得到一系列事实，如："元素 3 和元素 7 是同一类；元素 19 和元素 4 是同一类；元素 7 和元素 12 是同一类，…。"或者有一个过程，可以用来判断任何给定元素 $j$ 和 $k$ 是否属于同一类还是属于不同类。(回忆一下，一个等价关系可以是满足 **RST 性质**的任何关系，即满足自反性、对称性和传递性。这是和"同一性"的任何直观定义相兼容的。)

希望得到的输出是，将 $N$ 个元素中的每一个赋与一个等价类序号，当且仅当两个元素被赋与同一类序号时它们属于同一类。

有效的算法是这样进行的：令 $F(j)$是元素 $j$ 的类序号或"家族"序号，从每个元素在它自己家族开始，所以 $F(j)=j$。数组 $F(j)$能被解释成一个树结构，其中 $F(j)$代表 $j$ 的双亲。如果我们将每个家族排成它自己的树，与所有其它"家族树"不相交，那么，我们就可以用它最老最老的祖先来标志这一家族(等价类)。这一树的具体拓朴结构并不重要，只要我们在某处将有关的元素嫁接上去即可。

因此，我们按如下方式来处理每个基本的论据"$j$ 等价于 $k$"：(i) 追溯 $j$ 到它的最老的祖先，(ii) 追溯 $k$ 到它的最老的祖先，(iii) 令 $j$ 为 $k$ 的新的双亲，或反之亦然(这没有什么区别)。当处理所有的关系之后，我们检查所有的元素 $j$，并将它们的 $F(j)$重新置成其可能的最老的祖先，从而它标示了等价类。

下面是基于 Knuth[1]的程序，假定有 m 个基本信息。它们存储于两个长度为 m 的数组中 lista 和 listb。它的解释是：lista[j]和 listb[j](j=1,...,m)是两相关元素的序号(我们是这样被告知的)。

```
void eclass(int nf[], int n, int lista[], int listb[], int m)
    在输入数组 lista[1..m]和 listb[1..m]中，给出 n 个单个元素对之间 m 个两两等价信息，本程序返回 nf[1..n]标明
    每个元素的等价类序号，该序号是从1到 n 的整数(不是都要用到)。
{
    int l,k,j;

    for (k=1;k<=n;k++) nf[k]=k;                    将每个元素初始化成它自身的一类
    for (l=1;l<=m;l++){                             对每一条输入信息
        j=lista[l];
        while (nf[j] != j) j=nf[j];                 追溯第一个元素到它的祖先
        k=listb[l];
        while (nf[k] != k) k=nf[k];                 追溯第二个元素到它的祖先
        if (j != k) nf[j]=k;                        如果它们还没有相关，则使它们相关
    }
    for (j=1;j<=n;j++)                              最后上扫到最高祖先
        while (nf[j] != nf[nf[j]]) nf[j]=nf[nf[j]];
}
```

或有可能，我们可以构造一个函数 equiv(j,k)，它在 j 和 k 相关时返回一个非零值(正确)，或在不相关时返回零(错误)。于是，我们循环所有的元素对，就可得到一个完整的图片。D. Eardley 发明了一种聪明的方法，它能在完成上述要求的同时，以一种流畅的且能消除大多数最终扫视的方法，对树结构扫视到老祖先。

```
void eclazz(int nf[], int n, int (*equiv)(int, int))
    给出一个用户提供的逻辑函数 equiv，它告之一对元素是否相关，(每个元素都在范围1..n 中)。以 nf[1..n]返回每
    个元素的等价分类号。
{
    int kk,jj;

    nf[1]=1;
    for (jj=2;jj<=n;jj++) {                         对所有对的第一个元素循环
        nf[jj]=jj;
        for (kk=1;kk<=(jj-1);kk++){                 对所有对的第二个元素循环
            nf[kk]=nf[nf[kk]];                      上扫到这么多
            if ((*equiv)(jj,kk))   nf[nf[nf[kk]]]=jj;   让读者清楚为什么必须这样多的祖先是一个很
                                                    好的练习
        }
    }
    for (jj=1;jj<=n;jj++) nf[jj]=nf[nf[jj]];        最终只需要这么多的扫视
}
```

## 参考文献和进一步读物：


Knuth, D. E. 1968, *Fundamental Algorithms*, vol. 1 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), § 2.3.3. [1]

# 第九章　求根与非线性方程组

## 9.0　引言

现在,我们来考虑一个最基本的问题,即用数值方法求解方程的根。绝大多数方程既有左端项又有右端项,因此,传统上,人们常将两端相减,化为

$$f(x) = 0 \tag{9.0.1}$$

的形式以求其解,当仅有一个独立变量时,问题称为**一维的**,即求一元函数的一个或多个根。

如果独立变量不只一个,则有不至一个方程可以同时被满足。读者可能学过**隐函数理论**,这个理论告诉我们,具有 $N$ 个变量的 $N$ 个方程有希望被同时满足。注意,这里仅仅是希望而不是肯定。一个方程的非线性集合可能根本没有(实)解;相反,它也可能有多个解。隐函数理论告诉我们,"一般地",这若干个解之间应是互不相同、呈点状分布、并且相互分离的。但是,在极不凑巧的情况下可能会遇到非一般的,即退化的情况,这时又可能得到连续的一族解。如果将问题用向量记号表示,即求一个或多个 $N$ 维向量 $\mathbf{x}$,使得

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \tag{9.0.2}$$

这里 $\mathbf{f}$ 是 $N$ 维向量值的函数,由同时被满足的 $N$ 个独立方程组成。

读者不要被方程(9.0.1)与(9.0.2)形式上的相似所迷惑。寻求同时满足多个方程的 $N$ 维解,比一维情况下的求根要复杂得多。它们的主要区别在于,一维情况下可以确定解的区间,然后在解区间内"搜索"方程的根;而在多维场合,我们始终不能确定根在何处,直到确实找到它为止。

除线性情况外,求根过程总要借助迭代来完成,在一维和多维情况下都是如此。常用的算法总是从某个近似的初始解出发,不断修改这个解,直至满足某种预先确定的收敛准则为止。对于光滑变化的函数,如果初始预测解比较合适,则好的算法总是可以收敛的。实际上,绝大多数算法的收敛速度甚至都可以预先确定。

但是,也不能过分强调求解成功与否是如何依赖于初始预测解,特别是对于多维问题。这个具有决定意义的开始步骤通常依赖于分析,而不是单纯数值。精心选择的初始预测不仅可以减少计算量,而且可以增强理解和自信。汉明(Hamming)曾指出,"计算的目的在于洞察,而不仅仅限于数字",这个观点特别适于求根过程。当读者的程序以十位数字的精确度收敛于一个问题的错解,或者由于根本无解;或虽有解但因预测初值没有足够靠近该值而使程序不能收敛时,请仔细回味汉明的这句名言。

读者可能会问,"汉明的观点的确是高见,但是到底我应该怎样做呢?"对一维求根问题,回答很直截了当:在试着开始求解之前,应该在脑子里对方程的特点有一个大致的轮廓。如果为成批求解需要有很多的不同方程,那么至少应对其中一些典型方程做到心中有数。且

次，应确定好求解区间，即确定函数在什么区间内改变其正负号。

最后（对这一点某些大胆的读者可能有不同看法，但我们仍在此提出来），在迭代的每一步中，都不要让迭代算法跑到已获得所划定最佳搜索区间之外。下面我们将会看到，一些在教学上很重要的算法，如**弦截法**、**牛顿-拉斐森**（**Newton-Raphson**）**算法**等都可能会违背这条限制。因此，如果不对这些算法做某些改进，建议读者最好不要使用它们。

出现多重根或根相近的情况，的确是个令人头痛的问题，特别是当重数为偶数时。在这种情况下，函数可能不会有明显的符号变化，因而确定和保持解区间就变得很困难了。但是我们可以采取强硬措施，即硬行确定一个解区间，哪怕需要使用第十章中的极小值搜索技术，来决定函数的某个低峰是否确实通过零点。（读者可以将第10.1节中的简单黄金分割法程序做一点小的修改，使之它一旦检查出函数的符号变化，就立即返回调用处；而且可以判断如果函数的极小值恰好是零，则我们找到的一个二重根。）

我们照例不主张读者把了程序当作“黑箱”使用，而一概对其内部结构不求理解。但是，对初学者来说，以下几条建议是很可取的：

· 在函数的导数不易计算时，可以选用第9.3节中的布伦特(Brent)算法，它适于求解一般一元函数的根，但这个根应位于搜索区间内。第9.2节中里德(Ridder)的方法简明扼要，也是比较理想的选择。

· 如果函数的导数是可以计算的，建议读者利用第9.4节中程序 **rtsafe**，该程序把对边界信息所作的某些记录与牛顿－拉斐森方法结合起来，但它同样也需要首先确定解区间。

· 多项式求根是一种特殊情况，可以用第9.5节中的拉盖尔(Laguerre)方法作为起点。应注意的是，有些多项式是病态的。

· 最后，对于多维问题，除牛顿－拉斐森方法（第9.6节）外，本书将不再讨论其他算法。如果能给出一个很合理的关于解的初始预测，牛顿－拉斐森将是一个相当不错的算法，读者不妨试一试。此外第9.7节中，还有几种更复杂、但全局收敛性更好的算法，可供读者作进一步的阅读参考。

为适于不同的计算机实现，本书一般都避开了交互式的，或与图形相关的程序。但此处仍有一个例外。在下面的程序中，我们用交互式的方法确定坐标轴的刻度，从而产生一幅粗略的函数图形。对于初涉方程求根领域的读者，采用这种方法可以避免很多麻烦。

```
#include <stdio.h>
#define ISCR 60                    屏上所显示的水平和垂直位置的数目
#define JSCR 21
#define BLANK ' '
#define ZERO '-'
#define YY '1'
#define xx '-'
#define FF 'x'

void scrsho(float (*fx)(float))
    适用于交互式 CRT 终端，在区间 x1至 x2内绘出函数 fx 大致图形。等待输入另一图形区间，直到用户输入结束标
    志，以示满意为止。
{
    int jz,j,i;
    float ysml,ybig,x2,x1,x,dyj,dx,y[ISCR+1];
    char scr[ISCR+1][JSCR+1];

    for (;;) {
```

```
        printf("\nEnter x1 x2 (x1=x2 to stop):\n");        等待输入另一个图形，当
        scanf("%f %f",&x1,&x2);                               x1=x2 时退出
        if (x1 == x2) break;
        for (j=1;j<=JSCR;j++)                                 垂直边填入字符 y
            scr[1][j]=scr[ISCR][j]=YY;
        for (i=2;i<=(ISCR-1);i++) {
            scr[i][1]=scr[i][JSCR]=XX;                        上、下端填入字符 x
            for (j=2;j<=(JSCR-1);j++)                         内部填以空格
                scr[i][j]=BLANK;
        }
        dx=(x2-x1)/(ISCR-1);
        x=x1;
        ysml=ybig=0.0;                                        边界包括 0
        for (i=1;i<=ISCR;i++) {                               计算函数在等间隔点的值，找出最大与最小值
            y[i]=(*fx)(x);
            if (y[i] < ysml) ysml=y[i];
            if (y[i] > ybig) ybig=y[i];
            x += dx;
        }
        if (ybig == ysml) ybig=ysml+1.0;                      确保顶端和底端分开
        dyj=(JSCR-1)/(ybig-ysml);
        jz=1-(int) (ysml*dyj);                                指出哪一行对应零
        for (i=1;i<=ISCR;i++) {                               在函数高度和零位处设置标记
            scr[i][jz]=ZERO;
            j=1+(int) ((y[i]-ysml)*dyj);
            scr[i][j]=FF;
        }
        printf(" %10.3f ",ybig);
        for (i=1;i<=ISCR;i++) printf("%c",scr[i][JSCR]);
        printf("\n");
        for (j=(JSCR-1);j>=2;j--) {                           显示
            printf("%12s"," ");
            for (i=1;i<=ISCR;i++) printf("%c",scr[i][j]);
            printf("\n");
        }
        printf(" %10.3f ",ysml);
        for (i=1;i<=ISCR;i++) printf("%c",scr[i][1]);
        printf("\n");
        printf("%8s %10.3f %44s %10.3f\n"," ",x1," ",x2);
    }
}
```

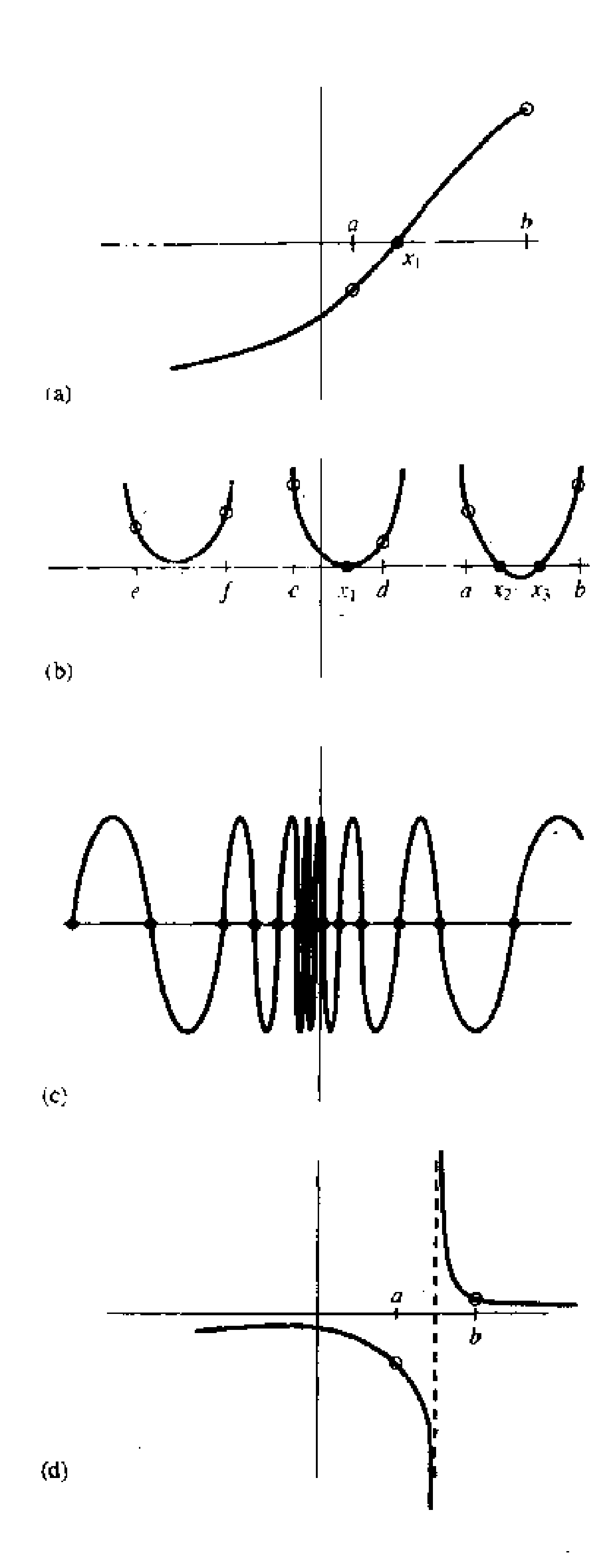
## 9.1 划界与二分

如果 $f(a)$ 与 $f(b)$ 具有相反的符号，则我们认为将有一根被划界在区间 $(a,b)$ 内，又若函数是连续的，则至少有一个根必落在该区间内（**中值定理**）。对非连续但有界的函数，则可能有一阶跃（而不是一个根）穿过零点（见图9.1.1）。从数值上解释，这一阶跃也可能是一个根，因为这种情况与某种连续函数很难区分，这种连续函数在机器的有限精度表示下，其零点可能会出现在两个“相邻”的浮点数之间。只有带奇点的函数才有可能在划定的区间内没有任何根存在。

例如，对于函数

$$f(x) = \frac{1}{x-c} \tag{9.1.1}$$

某些求根算法（如本节的二分法）可能会收敛到 $c$。幸运的是，大多数情况下几乎都不可能得到 $c$ 或任何接近于 $c$ 的错误解，因为只需稍稍计算一下 $|f(x)|$ 就会发现这个值很大而不是很小。




(a)表明由 $a$,$b$ 两点划界的一个孤立的根 $x_1$,其中 $a$,$b$ 两点函数值具有相反的符号;(b)说明在一个二重根附近,函数值不一定必须有符号变化(实际上也不一定必须有一个根!);(c)说明,一个病态函数可能有多个根;在(d)中,函数在 $a$,$b$ 两点符号相反,但 $a$,$b$ 两点之间包含的是一个奇异点,而不是一个根。

图9.1.1　在方程式求根过程中可能会遇到的几种情况。

如果函数是在黑箱中给出的，则没有一种确定的方法可以给根划界，甚至不能确定函数是否有根。例如病态方程(3.0.1)，它的两个实根仅仅落在一个极小的区间范围 $x=\pi+10^{-667}$ 内。

在下一章中，我们将讨论与本章相关的函数极小值的划界问题。对于这个问题，可以找到一种在任何情况下都能成功求解方法，从本质上讲，即"向下搜索，同时逐渐加大步长，直到函数值又开始回升为止"。但在求根过程中，则没有类似的方法可寻。"向下搜索，直到函数改变符号"的方法对于具有简单极值的函数并不适用。但是，如果我们对"失败"的结果有所准备，这种方法却常常是一个良好的开端：如果函数在 $x\to\pm\infty$ 的范围内具有相反的符号，则可以很有把握地得到成功的结果。

```
#include <math.h>
#define FACTOR 1.6
#define NTRY 50

int zbrac(float (*func)(float), float *x1, float *x2)
```

给定一个函数 func 及一个初始的预测范围 x1到 x2，本程序将这处范围进行几何扩展，直到方程有某个根被划界在返回的值 x1和 x2之间(此时，zbrac 返回1)，或者直至这个范围被扩展到大得不能接受的程度为止(这时 zbrac 返回0)。如果某个函数对于足够大和足够小的参数具有相反的符号，则本程序必能成功返回。

```
{
    void nrerror(char error_text[]);
    int j;
    float f1,f2;

    if (*x1 == *x2) nrerror("Bad initial range in zbrac");
    f1=(*func)(*x1);
    f2=(*func)(*x2);
    for (j=1;j<=NTRY;j++) {
        if (f1*f2 < 0.0) return 1;
        if (fabs(f1) < fabs(f2))
            f1=(*func)(*x1 += FACTOR*(*x1-*x2));
        else
            f2=(*func)(*x2 += FACTOR*(*x2-*x1));
    }
    return 0;
}
```

读者可能期望"向内"考察初始区间，而不希望将这个区间"向外"扩展，并且提出，如果将区间 $x_1$到 $x_2$划分为 $n$ 个等间隔的小区间来进行搜索，函数 $f(x)$在这些小区间内是否有根存在。下面的小程序可以计算出 $nb$ 个不同的小区间的边界，其中每个小区内包含一个或多个根。

```
void zbrak (float (*fx)(float), float x1, float x2, int n, float xb1[], float xb2[], int *nb)
```

给定一个定义在区间(x1,x2)内的函数 fx，将这个区间划分为 n 个等间隔的小段，并求函数过零点的次数。nb 的输入值表示搜索根的最大次数，它在返回时被重新赋值为所找到的划界对 xb1[1..nb]，xb2[1..nb]的个数。

```
{
    int nbb,i;
    float x,fp,fc,dx;

    nbb=0;
    dx=(x2-x1)/n;                    确定适当的网格划分
    fp=(*fx)(x=x1);
    for (i=1;i<=n;i++) {             对所有区间进行循环
```

```
        fc=(*fx)(x+=dx);
        if (fc*fp < 0.0) {                              若有符号变化，则记下边界值
            xb1[++(nbb)]=x-dx;
            xb2[nbb]=x;
            if (*nb == nbb) return;
        }
        fp=fc;
    }
    *nb=nbb;
}
```

### 9.1.1 二分法

一旦我们知道了某个区间内包含有方程的一个根，则有好几种经典方法可将该区间进一步细分。这些方法以不同的收敛速度和确定性程度趋于答案。但令人失望的是，能够保证收敛的方法求解速度却最慢，而那些收敛速度最快的方法却极有可能以极快的速度趋向无穷，而且如果没有采取避免这种情况的发生的话，算法本身不会给出任何警告信息。

**二分法**则是一种"常胜"的方法，其设计思想相当简单：如果在某一区间内函数的符号有变化，则在此区间内函数必过零。计算函数在该区间点的值，并考察其符号，用中点替代与具有相同函数符号的端点。每经过一次迭代，包含根的区间的就减小了一半，假设 $n$ 次迭代后，根位于长度为 $\epsilon_n$ 的区间内，则在下一轮迭代结束后，这个根将被划界在长度恰好为

$$\epsilon_{n+1} = \epsilon_n / 2 \tag{9.1.2}$$

的区间内，因此，我们便可以事先计算出要达到给定容限的解所需的迭代次数，为：

$$n = \log_2 \frac{\epsilon_0}{\epsilon} \tag{9.1.3}$$

这里 $\epsilon_n$ 的初始划分区间的长度，$\epsilon$ 为迭代结束时所期望达到的容限。

二分法必然求解成功。假如区间碰巧包含两个或多个根，则用二分法可以找到其中之一；如果区间内不包含方程的根而仅有一奇点，则二分法将收敛到该奇点。

如果一种方法的收敛，是以前次迭代不确定性的一阶幂乘以一个因子（小于1）的速度收敛，则称这种方法为**线性收敛**（如二分法就属于这种情况）。而以高阶幂收敛的方法，即

$$\epsilon_{n+1} = \text{常数} \times (\epsilon_n)^m \qquad m > 1 \tag{9.1.4}$$

称为超线性收敛。在其他书中可能会把"线性"收敛称作"指数"收敛或"几何"收敛，但名称是什么根本无所谓。线性收敛性就是指，有效数字随着计算而线性地逐次增加。

剩下的工作就是讨论如何确定一种实用收敛准则。机器中总是以固定位数的二进制数来表示浮点的，记住这一点极为关键。如果一个函数经分析可知它必过零点，但在计算中可能对任何浮点数变量，函数根本不取零值。因此，必须确定出方程的根可以达到什么样的精度值。例如，当方程根的值大小接近1，那么取收敛到绝对值小于 $10^{-6}$ 就是比较合理的；但这个精度对于接近 $10^{26}$ 的根当然是不可能达到的。由此，人们可能会想到确定一个相对的（分数的）收敛准则，但对接近于零的根来说，这个准则就又不适用了。为了最大限度地确保通用性，下面的程序要求用户指定一个绝对容限，使算法在区间的绝对长度小于该容限时终止迭代。通常，我们可能倾向于将该容限值取为 $\epsilon(x_1+x_2)/2$，其中 $\epsilon$ 为机器精度，$x_1$ 和 $x_2$ 为初始区间的两个端点。但当根位于零值附近时，必须注意仔细考虑什么样的容限值对我们的函数来说是合理的。下面程序对任何调用允许执行的最大迭代次数为 40 次，因为 $2^{-40} \approx 10^{-12}$。

```
#include <math.h>
#define JMAX 40                允许迭代的最大次数

float rtbis(float (*func)(float), float x1, float x2, float xacc)
    已知位于 x1和 x2之间的函数 func,用二分法求其根。在精度达到±xacc 之前,这个根将被不断修正,并最终以 rtbis 
    返回。
{
    void nrerror(char error_text[]);
    int j;
    float dx,f,fmid,xmid,rtb;

    f=(*func)(x1);
    fmid=(*func)(x2);
    if (f*fmid >= 0.0) nrerror("Root must be bracketed for bisection in rtbis");
    rtb = f < 0.0 ? (dx=x2-x1,x1) : (dx=x1-x2,x2);          确定搜索方向以使 f>0位于 x+dx 处
    for (j=1;j<=JMAX;j++) {
        fmid=(*func)(xmid=rtb+(dx *= 0.5));                 二分法循环
        if (fmid <= 0.0) rtb=xmid;
        if (fabs(dx) < xacc || fmid == 0.0) return rtb;
    }
    nrerror("Too many bisections in rtbis");
    return 0.0;                                              程序执行不到此处
}
```

## 9.2 弦截法、试位法和里德的方法

对于在根附近光滑的函数,**试位法**和**弦截法**的收敛速度通常要比二分法快。在这两种方法中,函数在所讨论的局部范围内均假设为近似线性,而且根的下一估计值总是取在接近直线与坐标轴的交点处。每一轮迭代结束后,上一轮迭代的边界点中将有一个被舍弃,以有利于根的最新估计。

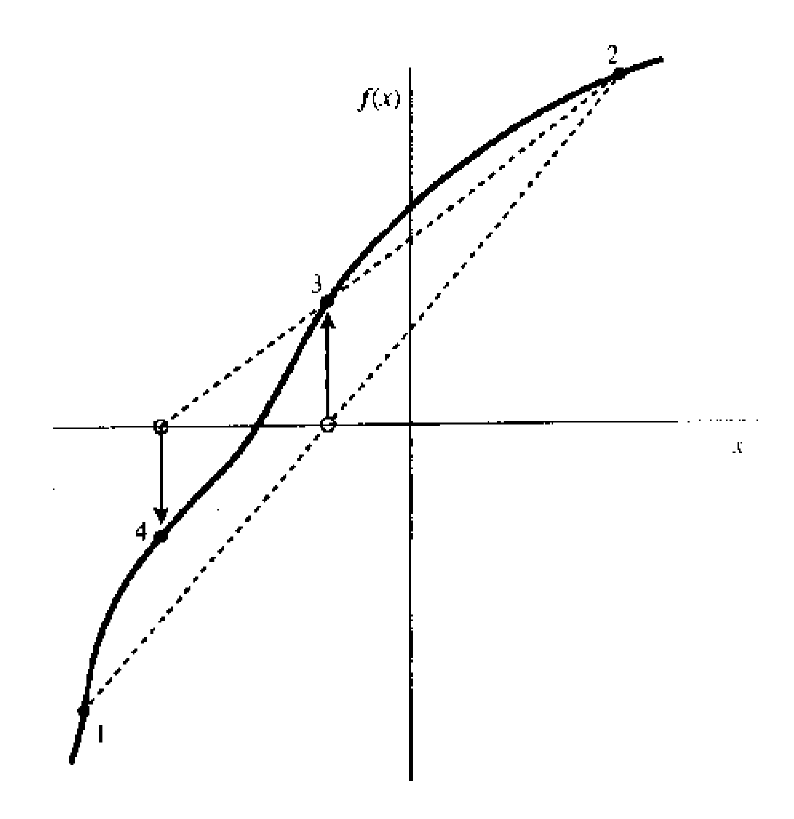
上述两种方法的唯一的区别在于,弦截法总是取两个端点的最新估计值(图9.2.1;该方法要求在第一轮迭代开始任意选择一个区间);而试位法需要保留上一轮迭代的一个端点,条件是该端点的函数值,与这一轮迭代求出的估计根的函数值,具有相反的符号,从而可以使这两点能继续划界根(图9.2.2)。精确地说,对于足够连续的函数,弦截法在根附近收敛速度加快,其收敛阶可达到"黄金比率" 1.618...,即

$$\lim_{k\to\infty}|\epsilon_{k+1}| \approx 常数 \times |\epsilon_k|^{1.618} \tag{9.2.1}$$

但是弦截法也有一大缺陷,即方程的根不一定总能落在迭代求出的划界之内。这样对于不是足够连续的函数,该算法就不能保证收敛,因为局部迭代过程可能会导致趋于无穷。

由于试位法有时需保留旧端点,而不取新计算出来的估计值。因而其收敛阶要稍低一些。正因为新的函数值只是有时被保留下来,所以这种方法常常是超线性收敛的,但是精确的收敛阶数却不易估算。

这两种相关方法的程序实现将在下文中给出。尽管它们都为教科书所采用,但程序后面给出的里德(Ridder)方法以及下节中将要介绍的 Breat(布伦特)方法常常更受青睐。图9.2.3展示的是弦截法和试位法对一个特殊函数的求根过程。

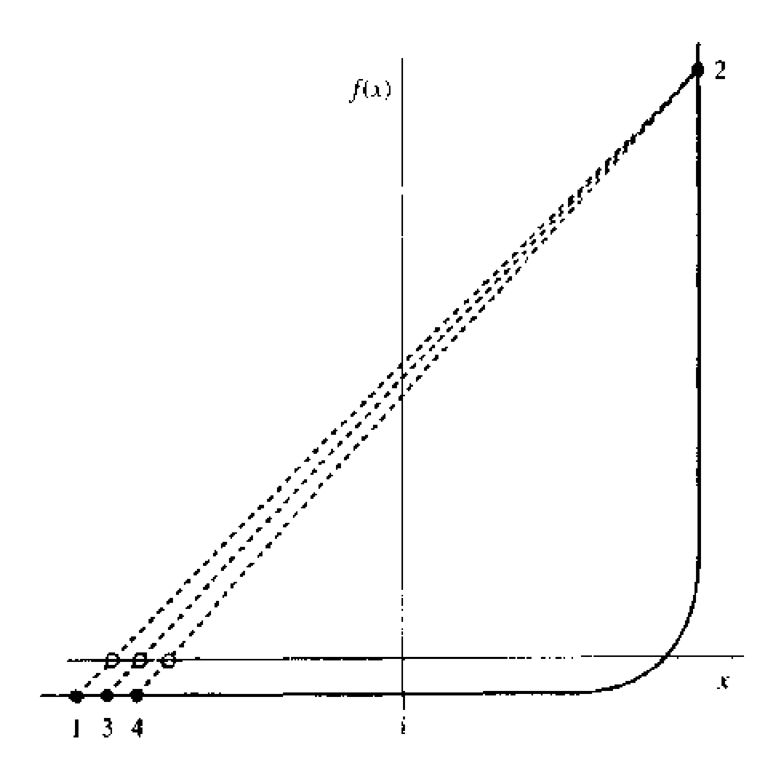



过最新计算出来的两点作内插线或外推线(虚线所示)而不论这两点是否划界这函数,图中的点是按其被使用的顺序进行编号的。

图9.2.1 弦截法图示

过划界根的最新端点作内插线(虚线所示),这两个端点之间必须包含有方程的根。在本例中,点1在很多步迭代中,都将保持“有效”而不被其他点取代。试位法比弦截法收敛速度慢,但比弦截法确保收敛的程度要高。

图9.2.2 试位法图示




对本图所示的函数，弦截法和试位法都需要进行很多步迭代才能求出真解，而用许多其他算法都不易求解.

图9.2.3

```
#include <math.h>
#define MAXIT 30                    允许迭代的最多次数

float rtflsp(float (*func)(float), float x1, float x2, float xacc)
```

已知位于 x1至 x2之间的函数 func，用试位法求其根，在精度达到±xacc 之前，这个根将不断地被修正，并最终以 rtflsp 返回。

```
{
    void nrerror(char error_text[]);
    int j;
    float fl,fh,xl,xh,swap,dx,del,f,rtf;

    fl=(*func)(x1);
    fh=(*func)(x2);                 确保区间内包含一个根
    if (fl*fh > 0.0) nrerror("Root must be bracketed in rtflsp");
    if (fl < 0.0) {                 调整端点标号，使x1对应于函数值的下限边
        xl=x1;
        xh=x2;
    } else {
        xl=x2;
        xh=x1;
        swap=fl;
        fl=fh;
        fh=swap;
    }
    dx=xh-xl;
    for (j=1;j<=MAXIT;j++) {        试位法循环
        rtf=xl+dx*fl/(fl-fh);       根据最新值调整增量
        f=(*func)(rtf);
```

```
        if (f < 0.0) {                  替代满足条件的端点
            del=xl-rtf;
            xl=rtf;
            fl=f;
        } else {
            del=xh-rtf;
            xh=rtf;
            fh=f;
        }
        dx=xh-xl;
        if (fabs(del) < xacc || f == 0.0) return rtf;       收敛
    }
    nrerror("Maximum number of iterations exceeded in rtflsp");
    return 0.0;                         程序执行不到此处
}
```

```
#include <math.h>
#define MAXIT 30                        允许迭代的最多次数

float rtsec(float (*func)(float), float x1, float x2, float xacc)
```

若函数 func 位于 x1至 x2之间,用弦截法求它的根。在精度达到±xacc 之前,不断地修正这个根,并最终以 rtsec 返回。

```
{
    void nrerror(char error_text[]);
    int j;
    float fl,f,dx,swap,xl,rts;

    fl=(*func)(x1);
    f=(*func)(x2);
    if (fabs(fl) < fabs(f)) {         将具有较小函数值的边界作为根的最新估值
        rts=x1;
        xl=x2;
        swap=fl;
        fl=f;
        f=swap;
    } else {
        xl=x1;
        rts=x2;
    }
    for (j=1;j<=MAXIT;j++) {          弦截法循环
        dx=(xl-rts)*f/(f-fl);         根据最新值调整增量
        xl=rts;
        fl=f;
        rts += dx;
        f=(*func)(rts);
        if (fabs(dx) < xacc || f == 0.0) return rts;    收敛
    }
    nrerror("Maximum number of iterations exceeded in rtsec");
    return 0.0;                       程序永远执行不到此处
}
```

## 9.2.1 里德方法

里德(Ridder)[1]对试位法作了极有益的改进,其基本思想是,当某个根位于 $x_1$ 与 $x_2$ 之间时,首先计算函数在中点 $x_3=(x_1+x_2)/2$ 的值,然后对下述指数函数进行因式分解,这个独特的函数将残差函数变为一条直线:

$$f(x_1)-2f(x_3)e^{Q}+f(x_2)e^{2Q}=0 \tag{9.2.2}$$

解这个关于 $e^{Q}$ 的二次方程,可得:

$$e^{Q}=\frac{f(x_3)+\mathrm{sign}[f(x_2)]\sqrt{f(x_3)^2-f(x_1)f(x_2)}}{f(x_2)} \tag{9.2.3}$$

接下来将试位法应用于 $f(x_1), f(x_3)e^Q, f(x_2)e^{2Q}$(注意不是 $f(x_1), f(x_3), f(x_2)$)三个值,得到关于根的一个新估计值 $x_4$。总的更新公式为:

$$x_4 = x_3 + (x_3 - x_1)\frac{\operatorname{sign}[f(x_1) - f(x_2)]f(x_3)}{\sqrt{f(x_3)^2 - f(x_1)f(x_2)}} \tag{9.2.4}$$

(这里利用了求解式(9.2.3)。)

式(9.2.4)具有一些非常好的性质:首先,$x_4$ 肯定落在区间$(x_1, x_2)$内,因此里德法在迭代过程中不会使根越出划界;其次,连续利用式(9.2.4)可得收敛阶为二次,也即式(9.1.4)中的 $m$ 为2。由于每用一次式(9.2.4)需进行两次函数值的计算,因此里德法实际的收敛为 $\sqrt{2}$,而不是2;但是每经过两次的函数值计算以后,迭代结果的有效数字位数就几乎翻倍,这也可以算是超线性收敛;第三,由于里德法采用了指数(或称比率)因子,而不是通过多项式技术(如拟合一条抛物线)来消除函数的"转弯",因此它可称得上是一种特别有用的算法。在可靠性和收敛速度两个方面,里德法均可与我们下面将要讨论的 Van Wijngaarden, Dekker 以及 Brent 方法相媲美。

```
#include <math.h>
#include "nrutil.h"
#define MAXIT 60
#define UNUSED (-1.11e30)

float zriddr(float (*func)(float), float x1, float x2, float xacc)
```

已知位于 x1和 x2之间的函数 func,用里德的方法求其根。这个根在精度达到 xacc 之前将不断地被修正,并最终以 zriddr 返回调用处。

```
{
    int j;
    float ans,fh,fl,fm,fnew,s,xh,xl,xm,xnew;

    fl=(*func)(x1);
    fh=(*func)(x2);
    if ((fl > 0.0 && fh < 0.0) || (fl < 0.0 && fh > 0.0)) {
        xl=x1;
        xh=x2;
        ans=UNUSED;                                   设置一个最不可能出现的值以简化下面程序
                                                          中的逻辑判断
        for (j=1;j<=MAXIT;j++) {
            xm=0.5*(xl+xh);
            fm=(*func)(xm);                           每次迭代中计算的两个函数值中的第一个
            s=sqrt(fm*fm-fl*fh);
            if (s == 0.0) return ans;
            xnew=xm+(xm-xl)*((fl >= fh ? 1.0 : -1.0)*fm/s);      修正公式
            if (fabs(xnew-ans) <= xacc) return ans;
            ans=xnew;
            fnew=(*func)(ans);                        每次迭代中计算的两个函数值中的第二个
            if (fnew == 0.0) return ans;
            if (SIGN(fm,fnew) != fm) {                在下一次迭代中保证根被划界
                xl=xm;
                fl=fm;
                xh=ans;
                fh=fnew;
            } else if (SIGN(fl,fnew) != fl) {
                xh=ans;
                fh=fnew;
            } else if (SIGN(fh,fnew) != fh) {
                xl=ans;
                fl=fnew;
```

```
            } else nrerror("never get here.");
          if (fabs(xh-xl) <= xacc) return ans;
        }
        nrerror("zriddr exceed maximum iterations");
    }
    else {
        if (fl == 0.0) return x1;
        if (fh == 0.0) return x2;
        nrerror("root must be bracketed in zriddr.");
    }
    return 0.0;                              永远执行不到此处
}
```

参考文献和进一步读物：


Ridders, C. J. F. 1979, *IEEE Transactions on Circuits and Systems*, vol. CAS-26, pp. 979—980. [1]


## 9.3 范·维金加登-德克尔-布伦特方法

虽然弦截法和试位法一般来讲比二分法收敛速度快，但是在实践中人们发现，对于病态函数却是二分法的收敛速度要快些，如锯齿形函数，不连续函数，甚至是二阶导数在根附近有突变的光滑函数等等。二分法总是将解区间对分一半，而弦截法和试位法有时要作很多次迭代才能把两个相距较远的端点向根所在的位置拉近。里德的方法虽然很好，但有时反被弄复杂，弄蠢了。究竟有没有一种方法，既能超线性收敛，又不失二分法的确定性呢？

答案是肯定的。我们可以对一个假设为超线性收敛的方法进行跟踪和记录，看它是否真正地按假定的方式收敛；如果不是的话，可以在迭代过程中穿插一些二分法的步骤，以保证这种方法至少具有线性收敛性。这类堪称上策的方法，需要注意记录迭代过程中的细节，并需仔细考虑舍入误差对指导策略的影响，而且还必须对算法何时收敛做到心中有数。

六十年代，阿姆斯特丹数学中心的范·维金加登，德克尔(Van Wijingaarden, Dekker)等学者成功地研究出一种能够实现上述要求的算法，而后该算法又由布伦特(Brent)作了改进[1]。为简单起见，我们称该算法的最后版本为**布伦特方法**。只要函数在包含根的初始区间内是可计算的，布伦特方法即可保证收敛(这个结论已由布伦特证明)。

布伦特方法将根的划界、二分法以及二次反插值方法相结合，从过零附近的邻域作迭代收敛。前面介绍的试位法和弦截法是在两个根的估值之间进行近似的线性运算，而二次反插值则用三个估值点拟合一个二次反函数($x$ 作为 $y$ 的二次函数)，该函数在 $y=0$处的值将作为根 $x$ 的下一估值。当然，对于根落在解区间之外的情形应该也有应急措施。布伦特方法都考虑了上述各种情况出现的可能性。假设三个点对为$[a,f(a)]$,$[b,f(b)]$,$[c,f(c)]$，则内插公式(参考方程(3.1.1))为

$$x=\frac{[y-f(a)][y-f(b)]c}{[f(c)-f(a)][f(c)-f(b)]}+\frac{[y-f(b)][y-f(c)]a}{[f(a)-f(b)][f(a)-f(c)]}+\frac{[y-f(c)][y-f(a)]b}{[f(b)-f(c)][f(b)-f(a)]}\tag{9.3.1}$$

置 $y=0$，则可求得根的下一估值为
$$x=b+P/Q\tag{9.3.2}$$
其中 $P$、$Q$ 的意义如下：若记
$$R\equiv f(b)/f(c)\quad S\equiv f(b)/f(a)\quad T\equiv f(a)/f(c)\tag{9.3.3}$$

则有 $$P = S[T(R-T)(c-b)-(1-R)(b-a)] \tag{9.3.4}$$

$$Q=(T-1)(R-1)(S-1) \tag{9.3.5}$$

实际计算时，$b$ 将作为当前迭代中根的最佳估计，$P/Q$ 应为一个"小"的修正项。二次函数方法仅适合于比较光滑的函数，它们在运行过程中，极有可能迭代出与根相差很远的估计值，而且当 $Q$ 很小而接近零时，在机器中有被零除以至溢出的危险。布伦特方法在设计时，通过保持对方程根的划界，并在做除法之前检查应在何处作内插，因而避免了这些问题的发生。当修正项 $P/Q$ 使根的估计值落在解区间之外，或者解区间长度的减小速度不够快时，布伦特方法将采用二分法的步骤，也就是说，在适当的步骤中，将二分法的收敛确定性与高阶方法收敛速度快的优点结合起来使用。对只知函数值（而不可求其导数或函数的形式）的一般一元方程的求根，我们建议读者选用这种方法。

```
#include <math.h>
#include "nrutil.h"
#define ITMAX 100                       允许迭代的最大次数
#define EPS 3.0e-8                      机器的浮点数精度

float zbrent(float (*func)(float), float x1, float x2, float tol)
    已知位于x1和x2之间的函数func，用布伦特方法求它的根，在其精度达到tol以前，这个根将不断地被修正，并最终
    以zbrent返回。
{
    int iter;
    float a=x1,b=x2,c=x2,d,e,min1,min2;
    float fa=(*func)(a),fb=(*func)(b),fc,p,q,r,s,tol1,xm;

    if ((fa > 0.0 && fb > 0.0) || (fa < 0.0 && fb < 0.0))
        nrerror("Root must be bracketed in zbrent");
    fc=fb;
    for (iter=1;iter<=ITMAX;iter++) {
        if ((fb > 0.0 && fc > 0.0) || (fb < 0.0 && fc < 0.0)) {
            c=a;                                对 a,b,c 更名并调整解区间 d
            fc=fa;
            e=d=b-a;
        }
        if (fabs(fc) < fabs(fb)) {
            a=b;
            b=c;
            c=a;
            fa=fb;
            fb=fc;
            fc=fa;
        }
        tol1=2.0*EPS*fabs(b)+0.5*tol;        收敛性检验
        xm=0.5*(c-b);
        if (fabs(xm) <= tol1 || fb == 0.0) return b;
        if (fabs(e) >= tol1 && fabs(fa) > fabs(fb)) {
            s=fb/fa;                          试作二次反内插
            if (a == c) {
                p=2.0*xm*s;
                q=1.0-s;
            } else {
                q=fa/fc;
                r=fb/fc;
                p=s*(2.0*xm*q*(q-r)-(b-a)*(r-1.0));
                q=(q-1.0)*(r-1.0)*(s-1.0);
            }
            if (p > 0.0) q = -q;              检查是否在解区间内
```

```
        p=fabs(p);
        min1=3.0*xm*q-fabs(tol1*q);
        min2=fabs(e*q);
        if (2.0*p < (min1 < min2 ? min1 : min2)) {
            e=d;                            可以做内插
            d=p/q;
        } else {
            d=xm;                           不能做内插，改用二分法
            e=d;
        }
    } else {                                界下降速度太慢，改用二分法
        d=xm;
        e=d;
    }
    a=b;                                    将最新估值赋给 a
    fa=fb;
    if (fabs(d) > tol1)                     计算新的试验解
        b += d;
    else
        b += SIGN(tol1,xm);
    fb=(*func)(b);
  }
  nrerror("Maximum number of iterations exceeded in zbrent");
  return 0.0;                               永远不执行到此
}
```

参考文献和进一步读物：


Brent, R. P. 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, NJ: Prentice-Hall), Chapters 3, 4. [1]


## 9.4 利用导数的牛顿-拉斐森算法

在所有一维的求根方法中，最著名的当首推**牛顿法**，也称**牛顿-拉斐森**(Newton-Raphson)**算法**。这种方法与前面几节所述方法的区别在于，它既要利用任意点 $x$ 处的函数值 $f(x)$，还需计算其导数值 $f'(x)$。从几何上来看，牛顿—拉斐森方法是将当前点 $x_i$ 处的切线延长，使之与横轴相交，然后置下一估计值 $x_{i+1}$ 为交点处的横坐标值(见图9.4.1)。

从代数上解释，该方法的推导，利用了我们所熟悉的在某点附近函数的泰勒级数展开，即

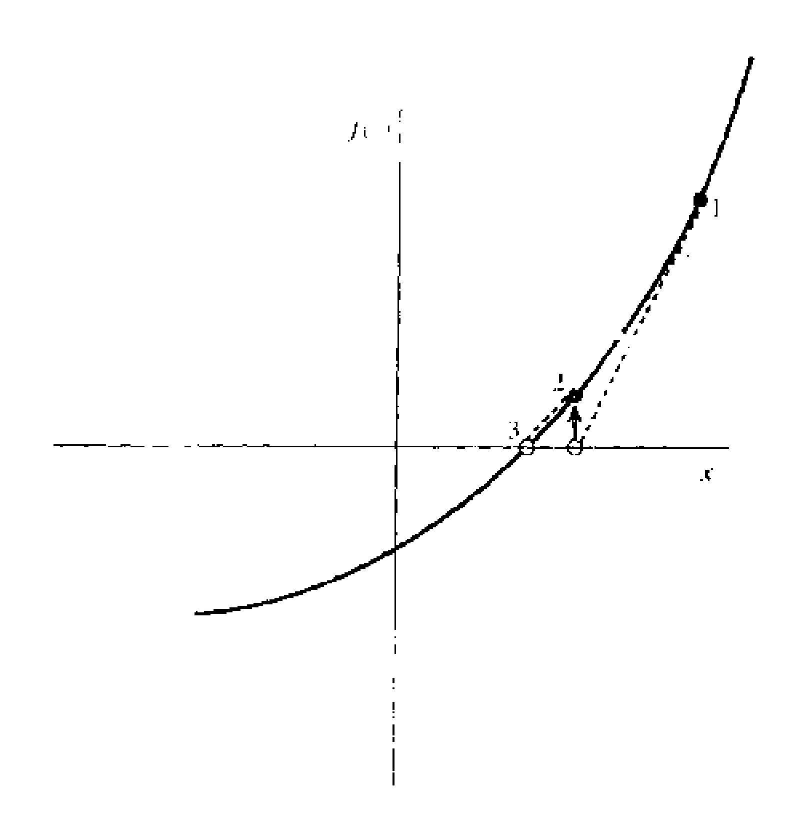
$$f(x+\delta)\approx f(x)+f'(x)\delta+\frac{f''(x)}{2}\delta^2+\ldots \tag{9.4.1}$$

对足够小的 $\delta$ 以及性能良好的函数，上式中二次及二次以上的项均可忽略不计，因此由 $f(x+\delta)=0$ 可导出

$$\delta=-\frac{f(x)}{f'(x)} \tag{9.4.2}$$

牛顿-拉斐森方法并不限于一维问题，正如我们在第9.6节以及第9.7节中将要看到的，它可推广至多维情形。

在离根较远的地方，展开式中的高阶项不能忽略，这时，牛顿-拉斐森方法可以粗略地给




对本图所示的函数，牛顿法的运行状态良好，结果达到了二次收敛。

图9.4.1　用牛顿方法将局部导数外推以寻求根的下一估计值

出一些非精确的而且没有什么特殊意义的修正项。例如，为了让搜索区间包含函数的某个局部极大值或极小值，根的初始预测可能需置在离真正的根很远的地方。这种情况有可能会导致求根失败（见图9.4.2）。如果某次迭代求出的估计值位于这样的局部极值附近，这时一阶导数几乎变为零，方程的根将“丢失”而且很难再找到。象绝大多数功能很强的算法工具一样，在不适当的场合，牛顿—拉斐森方法也可以是破坏性的。图9.4.3所示的是另外一种可能的病态情况。

为什么我们还要说牛顿-拉斐森是一种很有效的算法呢？原因在于其收敛的速率：在点 $x$ 的一个很小的邻域范围 $\varepsilon$ 内，函数及其导数可近似地表示为：

$$
\begin{aligned}
f(x+\varepsilon) &= f(x) + \varepsilon f'(x) + \varepsilon^2 \frac{f''(x)}{2} + \cdots, \\
f'(x+\varepsilon) &= f'(x) + \varepsilon f''(x) + \cdots
\end{aligned}
\tag{9.4.3}
$$

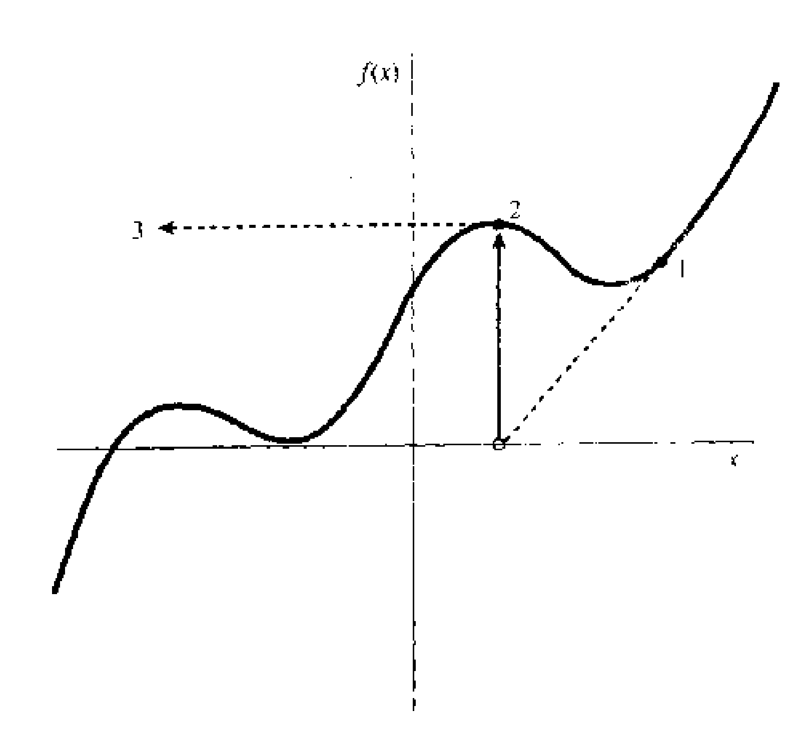
由牛顿-拉斐森公式，

$$
x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{9.4.4}
$$

有：

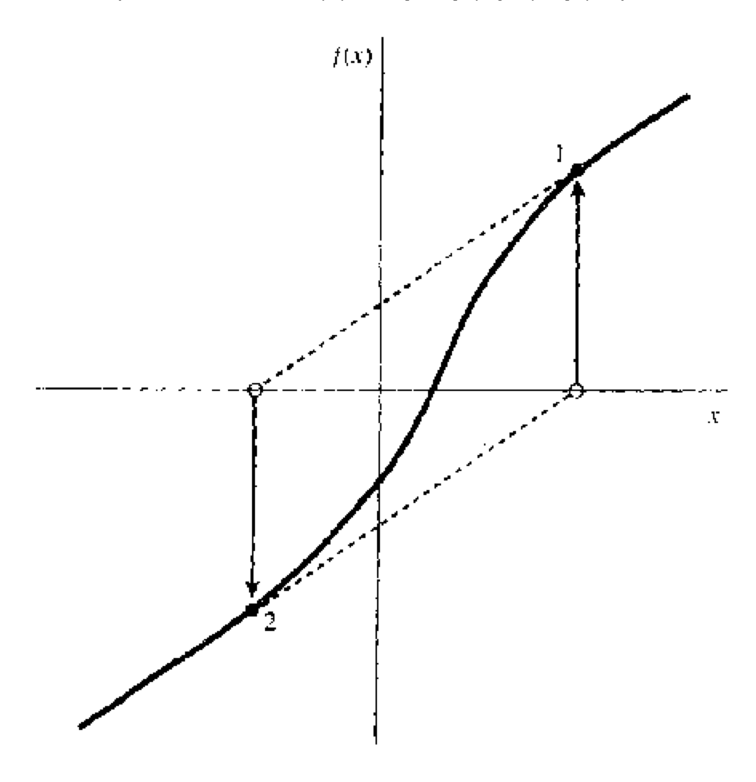
$$
\varepsilon_{i+1} = \varepsilon_i - \frac{f(x_i)}{f'(x_i)} \tag{9.4.5}
$$

如果估计值 $x_i$ 与真值相差 $\varepsilon_i$，则可利用式(9.4.3)，用 $\varepsilon_i$ 和根本身的导数来表示式(9.4.4)中的 $f(x_i)$ 及 $f'(x_i)$，这个结果是一个关于估计值导数的递推关系式，即




在迭代过程中遇到了局部极值，这时该点的切线将射向外部空间，永远无法与横轴相交，这里若用划界方法确定解区间(如 rtsafe 那样)就可以避免这种情况的发生。

图9.4.2 应用牛顿方法导致求根失败的示例




当函数 $f$ 全部或部分地由表插值得到时，常常会遇到这种情况，但若对初值预测得稍好一些，该例仍能求出真解

图9.4.3 应用牛顿方法导致死循环的示例

$$\epsilon_{i+1} = -\epsilon_i^2 \frac{f''(x)}{2f'(x)} \tag{9.4.6}$$

式(9.4.6)说明,牛顿-拉斐森是二次收敛算法(参考方程9.2.3)。在根附近,每迭代一步有效数字的位数几乎翻倍。这种很强的收敛性质,使得牛顿-拉斐森方法非常适合于那些导数易计算,并且在根附近有连续导数的函数。

即使在收敛的最初几步中,牛顿-拉斐森方法就被否定了(由于其弱的整体收敛性质),但是人们却经常在根的修正过程中,还穿插一、两步这种方法,因为它可以使有效数字的位数两倍或四倍地增加。

实现牛顿-拉斐森方法时需由用户提供一个子程序,用以计算函数及其一阶导数在任意点 $x$ 处的值 $f(x)$ 和 $f'(x)$。牛顿-拉斐森公式还有一个应用,即以一个数值差分来近似真值的局部导数:

$$f'(x) \approx \frac{f(x+dx)-f(x)}{dx} \tag{9.4.7}$$

但是,我们并不提倡这种方法,因为:(i) 每步要做两次函数值的计算,因此,超线性收敛的阶最多只能达到 $\sqrt{2}$ 。(ii) 如果 $dx$ 取得太小就会因四舍五入而被略去;而如果取得太大,收敛的阶又只能达到线性量级,这样还不如在随后的所有步骤中,都采用最初计算出来的 $f'(x_0)$。因此,利用数值导数的牛顿-拉斐森方法(在一维情况)通常要让位于第9.2节中的弦截法。(在多维情况,由于目前提供的方法很少,对待利用导数的牛顿—拉斐森方法需要更加慎重。详见第9.6节至第9.7节)。

下面的程序中调用一个由用户提供的函数 funcd(x,fn,df),其中 fn 为函数值、df 为导数值。这里我们包含一个对根的输入区间,目的只有为了与前面的程序相一致。牛顿方法不是对这个区间作调整,而只用到点 $x$ 附近的局部信息。这个输入区间只有两个用途:一是选它的中点作为迭代第一步的预测值;二是用于对跑到区间外的解,做出放弃的决定。

```
#include <math.h>
#define JMAX 20                     设置最大迭代次数

float rtnewt(void (*funcd)(float, float *, flaot *), float x1, float x2, float xacc)
    已知位于区间(x1,x2)内的一个函数,用牛顿—拉斐森方法求它的根。这个根 rtnewt 将不断地被修正,直到其精度
    达到±xacc 内为止。funcd 是一个用户提供的子程序,求在任意点 x 处的函数值及其一阶导数值。
{
    void nrerror(char error_text[]);
    int j;
    float df,dx,f,rtn;

    rtn=0.5*(x1+x2);                              初始预测
    for (j=1;j<=JMAX;j++) {
        (*funcd)(rtn,&f,&df);
        dx=f/df;
        rtn -= dx;
        if ((x1-rtn)*(rtn-x2) < 0.0)
            nrerror("Jumped out of brackets in rtnewt");
        if (fabs(dx) < xacc) return rtn;          收敛
    }
    nrerror("Maximum number of iterations exceeded in rtnewt");
```

```
    return 0.0;                                永不执行到此处
}
```

虽然牛顿-拉斐森方法的整体收敛性不够好，但是将它与二分法结合，设计一个失效保护程序却相对较容易。这种混合算法的基本思想是，一旦牛顿-拉斐森法在运行过程中出现解跑出解区间或者解区间长度的减小速度不够快的情况，就即刻采取一步二分法。

```
#include <math.h>
#define MAXIT 100                              允许最大迭代次数

float rtsafe(void (*funcd)(float, float *, float *), flaot x1, float x2, float xacc)
```

用牛顿—拉斐森和二分法相结合的方法。求一个方程的根，这个根位于解区间(x1,x2)内，在其精度达到±xacc之前，这个根将不断地被修正，并最终以 rtsafe 返回。funcd 是一个由用户提供的程序，返回函数值及函数的一阶导数值。

```
{
    void nrerror(char error_text[]);
    int j;
    float df,dx,dxold,f,fh,fl;
    float temp,xh,xl,rts;

    (*funcd)(x1,&fl,&df);
    (*funcd)(x2,&fh,&df);
    if ((fl > 0.0 && fh > 0.0) || (fl < 0.0 && fh < 0.0))
        nrerror("Root must be bracketed in rtsafe");
    if (fl == 0.0) return x1;
    if (fh == 0.0) return x2;
    if (fl < 0.0) {                              确定搜索方向以使 f(xl) < 0.
        xl=x1;
        xh=x2;
    } else {
        xh=x1;
        xl=x2;
    }
    rts=0.5*(x1+x2);                             初始化根的估值、倒数第二步
    dxold=fabs(x2-x1);                           的步长以及最后一步的步长
    dx=dxold;
    (*funcd)(rts,&f,&df);
    for (j=1;j<=MAXIT;j++) {                     在允许迭代次数内循环
        if ((((rts-xh)*df-f)*((rts-xl)*df-f) >= 0.0)    若牛顿法超出解区间或收敛速
            || (fabs(2.0*f) > fabs(dxold*df))) {         度不够快，则采用二分法
            dxold=dx;
            dx=0.5*(xh-xl);
            rts=xl+dx;
            if (xl == rts) return rts;           根的变化很小可忽略
        } else {                                 可以采用牛顿法，取牛顿步长！
            dxold=dx;
            dx=f/df;
            temp=rts;
            rts -= dx;
            if (temp == rts) return rts;
        }
        if (fabs(dx) < xacc) return rts;         收敛准则
        (*funcd)(rts,&f,&df);
        每迭代一次都要重新计算函数及其导数值 ration.
        if (f < 0.0)                             使根保持在解区内
            xl=rts;
        else
            xh=rts;
    }
```

```
    nrerror("Maximum number of iterations exceeded in rtsafe");
    return 0.0;                                 永远不执行到此
}
```

对很多函数来说，导数 $f'(x)$ 常常先于函数 $f(x)$ 收敛到机器精度，在这种情况下不需要连续更新 $f'(x)$。我们建议读者在对函数的一个般性质了如指掌后，再采用这个小"技巧"；但如果导数计算很繁琐，这种方法倒是可以加快计算速度。(形式上，这只能使收敛达到线性量级，但是若导数无论如何都不变化的话，收敛性也仅如此。)

### 9.4.1 牛顿-拉斐森和分形

在我们反复提醒读者，在牛顿-拉斐森方法的全局收敛性无法预知的同时(尽管其局部收敛非常迅速)，探讨一下某些特殊方程迭代的初值集合也别有一番情趣，从这个初值集合出发，算法既可能收敛到方程的根，也有可能收敛不到方程的根。

考虑如下这个简单的方程：

$$z^3 - 1 = 0 \tag{9.4.8}$$

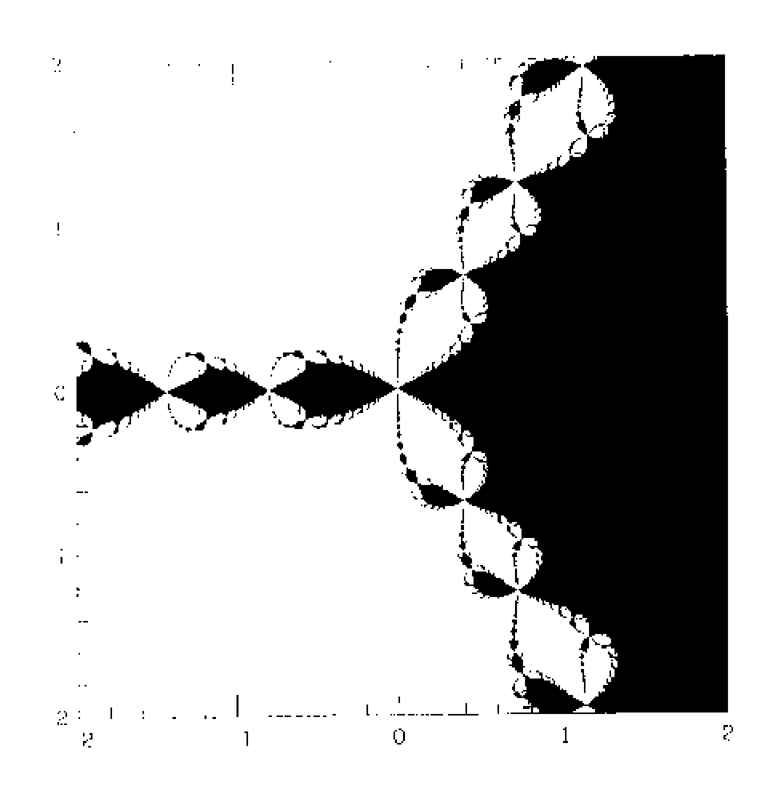
它的实数根只有一个即 $z=1$；其另外两个复数根是单位立方根，分别为 $\exp(\pm 2\pi i/3)$。由牛顿法导出的迭代公式为

$$z_{j+1} = z_j - \frac{z_j^3 - 1}{3z_j^2} \tag{9.4.9}$$

迄今为止，我们仅仅是将式(9.4.9)这样的迭代方程作用于实的初始值 $z_0$，但实际上这一节中所有方程都可应用于复平面。这样，我们就可以将复平面划分为一些区域，从这些区域出发，对某个初值 $z_0$ 用式(9.4.9)进行迭代，其结果或不收敛，或收敛到 $z=1$。我们可能会很自然地希望找到一个"收敛域"，该"收敛域"以某种方式包围 $z=1$ 这个根；当然"收敛域"不可能填满整个平面，因为平面内还必须包含收敛到两个复根的区域。实际上根据对称性，这三个区域应该具有相同的形状。(可能是三个对称的120°楔形，第个楔形的中心有一个根？)

图9.4.4展示的是一次经数值外推的结果。"收敛域"的确覆盖整个复平面的1/3区域，但它的边界可以说是极其不规则的，更确地讲是呈分形(fractal)(所谓分形就是它具有自身相似的结构，而且这种结构又以各种大小比例连续排列成形)。如此高度复杂的分形是如何从牛顿方法以及式(9.4.8)这样简单的方程中演变而来的呢？其实问题的答案早已隐含在图9.4.2中了，因为该图展示了某个局部极小点是如何导致牛顿法趋向于无穷远处的。设想我们从这个极小点处稍微挪开一段极小的距离，那么等待我们的将不会再是无穷远处的外部空间。如果幸运的话，我们也许能够恰好进入包含所求根的"收敛域"。但这意味着在这个极值的邻域内，必须存在一个"收敛域"的微型"复制品"，这个"复制品"可能产生了部分变形，具有某种"一级反射"的性质。同理可以推断，后面的迭代过程还会产生"二级反射"、"三级反射"…等等诸如此类的近似"收敛域"的微型复制品。这就是为什么会出现分形的原因。

注意到对方程(9.4.8)来说，几乎整个实轴都处在根 $z=1$ 的收敛域内。我们之所以对"整个实轴"冠以"几乎"二字，是因为位于实负半轴上的一些特殊离散点，它们的收敛性是不确定的(见图9.4.4)。如果从它们当中的某点开始执行牛顿方法，结果会如何呢？读者不妨试试看。



图中显示的是实轴和虚轴均在(−2,2)范围内的复平面 $z$,其中黑色区域为某些具有特殊性质的点的集合,从这些点出发,牛顿法将收敛到方程 $z^3-1=0$ 的根 $z=1$。黑色区域的形状就是所谓的分形。

图9.4.4

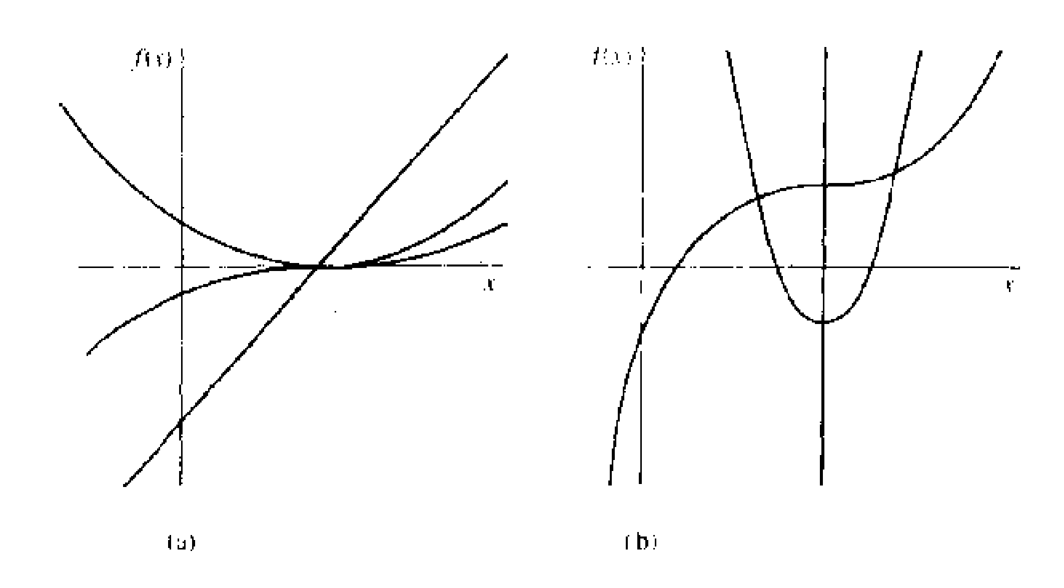
## 9.5 多项式的根

本节中我们将介绍几种多项式求根的方法。这些方法适用于求解绝大多数实际问题,包括阶数不太高的多项式以及性能良好的较高阶多项式的求根问题等。但不容乐观的是,仍有一些多项式的性质极差,而且在最坏情况下,甚至多项式系数的最小扰动都可能导致多项式的各个根零乱地散布到整个复平面上。(威尔金森(Wilkinson)构造了一个这样的例子,详阅阿克顿(Acton)的书[1]。)

我们知道,$n$ 阶多项式有 $n$ 个根,这些根可以是实的也可以是复的,而且它们当中可能有重根。如果多项式系数是实数,则它的复数根以共轭对的形式出现,也就是说,若 $x_1=a+bi$ 是该多项式一个根,则 $x_2=a-bi$ 必然也是其根。对系数是复数的多项式,其复根之间就不一定有关联了。

多重根或根相近的情况给数值算法带来极大的不便(见图9.5.1)。例如,$P(x)=(x-a)^2$ 在 $x=a$ 处有一个二重实根,但我们既不能用通常的方法确定解区间,即判断函数在哪些邻域内改变符号,也不能用象牛顿一拉斐森那样的斜率跟踪法,因为在重根处函数及其导数都将变为零。由于可能会出现大的舍入误差,牛顿-拉斐森方法即使是可行的,其速度也相当慢。如果预先能知道某个根是重根,则可以毫不费力地设计出特殊的处理方法。但在通常情

况下，我们并不能预先判断出根将会出现什么样的病态情况，这样便产生了各种各样的问题。




(a) 线性、二次及三次多项式在其根处的情况。(b) 只有将(a)高倍率放大才可以清楚地看到，三次多项式只有一个根而不是三个根；而二次多项式有两个根，而不是没有根。

图9.5.1 多项式的项

## 9.5.1 多项式的降阶

在求多项式的某几个根或所有根时，使用**降阶**方法可大大减少计算量。每求得一个多项式的根 $r$，我们便可以将该多项式因式分解为一个含根 $r$ 的项乘以一个比原多项式低一阶的多项式，即 $P(x)=(x-r)Q(x)$。由于 $Q(x)$ 的根是 $P(x)$ 根的子集，因此求根的计算量减小了；在相继求出 $P(x)$ 各根的同时，多项式的阶数也越来越低。更重要的是，用降阶方法可以避免一个严重错误，即可避免迭代算法两次收敛到同一个根，而这个根并不是重根，因为这种情况下正确的结果应是分别收敛到两个不同的根。

相当于综合除法的降阶方法可以看成是对多项式系数阵列的简单操作。在前面的第5.3节中曾给出了通过单项式因子进行综合除法的源程序代码。对复数根的降价，可以把该程序改写为适合于复数的类型；对实系数多项式但可能有复根的情况，也可以通过一个二次因子来降阶，即：

$$[x-(a+ib)][x-(a-ib)]=x^2-2ax+(a^2+b^2) \qquad (9.5.1)$$

第5.3节的程序 **poldiv** 可用于多项式除以上述因子。

但是，对降阶方法必须谨慎使用。因为新求出的每个根都只有有限的精确度，在确定逐次降阶的多项式的系数时就不知不觉地产生了累计的误差，结果会造成求出的根越来越不精确。无论这种不精确度是稳定地增长(即每步增加或减少机器精度的若干倍)，还是不稳定地增长(即有效数字不断减少，直到结果变得毫无意义)都是关系重大的。究竟哪一种情况发生仅仅取决于根是怎样被约去的。向前降阶在第5.3节中已经介绍过，这种方式中新多项式

系数的计算，是从 $x$ 的最高幂项开始顺序下去直到常数项。可以证明，如果在每一步中总是绝对值最小的根被约去，则这种方法是稳定的。与之相反的向后降阶在计算新多项式的系数时，是从常数项上升至 $x$ 的最高次幂项。如果每步中总是绝对值最大的根被约去，则向后降阶的方法也是稳定的。

如果把一个多项式的系数"尾—尾"交换(即把常数项变成最高幂项系数，等等)，则该多项式的根将被映射为其倒数。(证明：将整个多项式被其最高阶 $x^n$ 除，然后以 $1/x$ 为变量将结果重写为一个新的多项式的形式)。因此，向后降阶算法与向前降阶法实际上是等同的，只是原多项式的系数本身就是逆序的，并且使用了降阶根的倒数。因为下面我们将要使用向前降阶，所以向后降阶的算法将留给读者作为练习(和第5.3节中一样)。有关降阶方法稳定性的进一步探讨，请参考[2]。

使用降阶方法时，为了使累积误差(包括稳定的误差)的影响达到最小，不妨把逐次降阶后的多项式的根看作是原多项式的试验根，然后以这些试验根作为初始预测值并进行修正，其中这些初始值将利用没有降阶的原多项式 $p$ 重新求解。这里同样需要特别小心，以免用降阶法求出的两个根因精度过低而导致被修正后都收敛到同一个根(这个根可不通过降阶求得)。这时会出现伪重根和丢根的情况，但所幸的是，我们可以对这种情况进行监测，即对每一个修正过的根与前面的不同试探根作相等比较。一旦上述情况发生，不妨对该多项式只作一次降阶(而且只对这个根)，然后再修正这个试探根；我们也可以通过马赫里(Maehly)方法(见下面的方程(9.2.29))来解决。

下面，我们将进一步讨论有关修正实根和共轭复根的技巧。首先，我们来看看一般策略。

实系数多项式的求根方法可分为两大类。一类是，设法获得最容易获得的目标，即相异的实根，在求根过程中采用与一般函数求根法基本相同的方法(前节已讨论过)，也就是先用试位方式确定解区间，然后，再利用比较保险的牛顿—拉斐森方法(如程序 **rtsafe**)。有些情况下，我们可能只对多项式的实数根感兴趣，这时第一类方法就足以应付自如了。实系数多项式求根方法的第二类方法是，利用形如式(9.5.1)的二次因子。贝尔斯托(Bairstow)方法就属于这一类，我们将在下文有关根的修正一节中进行讨论；另外还有一种由米勒(Muller)提出的方法，这里简要介绍如下：

### 9.5.2 米勒的方法

**米勒**(Muller)**的方法**是弦截法的推广，但它采用的是三点之间的二次内插，而不是两点间的线性内插。此外二次式求根允许用此算法求解复根对。考给定根的三个估计值 $x_{i-2}$，$x_{i-1}$，$x_i$ 及多项式 $P(x)$ 在这些点处的值，则根的下一估计值 $x_{i+1}$ 可由下列公式计算得到：

$$
\begin{aligned}
q &\equiv \frac{x_i - x_{i-1}}{x_{i-1} - x_{i-2}} \\
A &\equiv qP(x_i) - q(1+q)P(x_{i-1}) + q^2P(x_{i-2}) \\
B &\equiv (2q+1)P(x_i) - (1+q)^2P(x_{i-1}) + q^2P(x_{i-2}) \\
C &\equiv (1+q)P(x_i)
\end{aligned}
\tag{9.5.2}
$$

则有

$$
x_{i+1} = x_i - (x_i - x_{i-1})\left[\frac{2C}{B \pm \sqrt{B^2 - 4AC}}\right] \tag{9.5.3}
$$

上式中分母内正负号的选取原则是使分母的绝对值或模尽可能地大。迭代时初始值可以随意选取三个数，例如取实轴上等间隔的三个点。值得注意的是，米勒方法在实现过程中，必须允许复数分母的出现以及随之带来的复数运算。

有时候米勒方法也被用于在复平面上求解解析函数（不仅限于多项式）的复根，如在IMSL程序ZANLY[3]中。

### 9.5.3 拉盖尔(Laguerre)方法

本书所采用的方法属于实系数多项式求根方法的第二类。这类算法的数目极少，而且收敛速度有快有慢，但每种算法都可以收敛到各种类型的根：实根、复根、单根以及重根。我们可以先用这类算法求得 $n$ 阶多项式所有 $n$ 个根的试验值，然后再回过头来一一修正这些值。

在这些一般的复数算法中，拉盖尔方法是迄今为止最直接的方法。尽管它甚至在收敛到实根的情况下也需要进行复数运算，但对于所有根都是实数的多项式来讲，从任意初始点开始它都能确保收敛到某个根。然而，对于具有复根的多项式，有关拉盖尔方法的收敛性证明在理论上几乎还是零。但不少经验表明，不收敛的情况出现的次数极小，而且几乎总是可以通过冲破某个有限的非收敛圈来解决。（这种解决方法在下面的程序中有具体的实现步骤。）例如，对于一个高阶（$\gtrsim 20$）多项式的示例，它的全部根恰好位于复数单位圆之外，并且基本上呈等距离分布；当算法收敛到零这个简单的复根时，可知其收敛阶为三次项。

在某些情况下，由于多项式本身可能具有复数系数，因此拉盖尔方法对复数运算并不会带来什么不便。

为推导拉盖尔(Laguerre)公式（虽然不是严格的推导），我们记多项式与其根及导数的关系如下所示：

$$P_n(x) = (x - x_1)(x - x_2)\ldots(x - x_n) \tag{9.5.4}$$

$$\ln|P_n(x)| = \ln|x - x_1| + \ln|x - x_2| + \ldots + \ln|x - x_n| \tag{9.5.5}$$

$$\frac{d\ln|P_n(x)|}{dx} = + \frac{1}{x - x_1} + \frac{1}{x - x_2} + \ldots + \frac{1}{x - x_n} = \frac{P_n'}{P_n} \equiv G \tag{9.5.6}$$

$$\frac{d^2\ln|P_n(x)|}{dx^2} = + \frac{1}{(x - x_1)^2} + \frac{1}{(x - x_2)^2} + \ldots + \frac{1}{(x - x_n)^2}$$

$$= \left[\frac{P_n'}{P_n}\right]^2 - \frac{P_n''}{P_n} \equiv H \tag{9.5.7}$$

由这些关系式开始，拉盖尔构造了"一个非常有力的假设集"（阿克顿(Acton)称之，见[1]）：假设所求的根 $x_1$ 与当前估计值 $x$ 的距离为 $a$，而其他根与 $x$ 的距离为 $b$，即

$$x - x_1 = a \quad ; \quad x - x_i = b \quad i = 2,3,\ldots,n \tag{9.5.8}$$

则式(9.5.6)及(9.5.7)可分别表示为

$$\frac{1}{a} + \frac{n-1}{b} = G \tag{9.5.9}$$

$$\frac{1}{a^2} + \frac{n-1}{b^2} = H \tag{9.5.10}$$

从而解得 $a$ 的表达式为

$$a = \frac{n}{G \pm \sqrt{(n-1)(nH - G^2)}} \tag{9.5.11}$$

这里正负号的选取要使分母的数值达到最大。由于上式中平方根一项有可能是负的，因此 $a$ 可以为复数（有关式(9.5.11)的严格推导，参见[4]）。

拉盖尔方法是以迭代方式进行的，即：对某个试验值 $x$，用式(9.5.11)计算 $a$，以 $x-a$ 作为下一个试验值，重复以上步骤直到 $a$ 足够小为止。

下面是拉盖尔方法的程序实现，该程序用来求解某个给定的 m 阶复系数多项式的一个根。这里和前面一样，第一个 a[0]将用于存储常数项，而 a[m]则存放 x 的最高阶项系数。拉盖尔算法采用简化的亚当斯(Adams，[5])终止准则，该准则在以下两方面作了巧妙的权衡，即一方面要保证足够的机器精度，另一方面又要考虑由舍入误差引起的无穷迭代的危险性。

```
#include <math.h>
#include "complex.h"
#include "nrutil.h"
#define EPSS 1.0e-7
#define MR 8
#define MT 10
#define MAXIT (MT * MR)
```

EPSS 为预先给定的舍入误差，在 MT 步的每一步中我们以 MR 个不同的分数值冲破限制圈，这里允许迭代的最多次数为 MAXIT。

```
void laguer(fcomplex a[], int m, fcomplex *x, int *its)
```

给定多项式 $\sum_{i=0}^{m} a[i]x^i$ 的阶数 m 及其 $m+1$ 个复系数 a[0..m]，同时给定一个复数 $x$，本程序将用拉盖尔方法不断修正这个 x，直到它在可以达到的舍入极限范围内，收敛到多项式的某个根为止。所经迭代的次数将以 its 返回。

```
{
    int iter,j;
    float abx,abp,abm,err;
    fcomplex dx,x1,b,d,f,g,h,sq,gp,gm,g2;
    static float frac[MR+1] = {0.0,0.5,0.25,0.75,0.13,0.38,0.62,0.88,1.0};
                                                    分数 frac 用于冲破某个限制圈
    for (iter=1;iter<=MAXIT;iter++) {               在允许的迭代次数范围内做循环迭代
        *its=iter;
        b=a[m];
        err=Cabs(b);
        d=f=Complex(0.0,0.0);
        abx=Cabs(*x);
        for (j=m-1;j>=0;j--) {                      多项式及其两个一阶导数的快速计算
            f=Cadd(Cmul(*x,f),d);
            d=Cadd(Cmul(*x,d),b);
            b=Cadd(Cmul(*x,b),a[j]);
            err=Cabs(b)+abx*err;
        }
        err *= EPSS;                                计算多项式值时估计舍入误差
        if (Cabs(b) <= err) return;                 已找到根，返回
        g=Cdiv(d,b);                                一般情形：利用拉盖尔公式
        g2=Cmul(g,g);
        h=Csub(g2,RCmul(2.0,Cdiv(f,b)));
        sq=Csqrt(RCmul((float) (m-1),Csub(RCmul((float) m,h),g2)));
        gp=Cadd(g,sq);
        gm=Csub(g,sq);
        abp=Cabs(gp);
        abm=Cabs(gm);
        if (abp < abm) gp=gm;
        dx=((FMAX(abp,abm) > 0.0 ? Cdiv(Complex((float) m,0.0),gp)
            : RCmul(exp(log(1+abx)),Complex(cos((float)iter),sin((float)iter)))));
        x1=Csub(*x,dx);
```

```
        if(x->r == x1.r && x->i == x1.i) return;           已收敛
        if(iter%MT) *x=x1;
        else *x=Csub(*x,RCmul(frac[iter/MT],dx));          偶尔采取一次冲破限制圈的步骤
    }                                                      (这种情况极少发生)
    nrerror("too many iterations in laguer");              根不常见——只有复根才可能发生。不妨改
    return;                                                变一下初值
}
```

下面的驱动程序通过连续调用函数 **laguer** 求解多项式的每一个根。该程序首先对多项式进行降阶,然后以同样的拉盖尔方法(如果不打算用其他方法的话)随意地对根作修正,最后将它们按其实部排序(在第13章中我们将利用这个驱动程序)。

```
#include <math.h>
#include "complex.h"
#define EPS 2.0e-6
#define MAXM 100                    m的最大允许值,为一个较小数

void zroots(fcomplex a[],int m,fcomplex roots[],int polish)
    给定多项式 ∑_{i=0}^{m} a[i]x^i 的阶数 m 及 m+1个复系数 a[0..m],本程序通过连续调用 laguer 以求解多项式的所有
    m 个复根,并将它们存储在数组 roots[1..m]中,如果根需要修正,(仍用拉盖尔方法),则逻辑变量 polish 的输入值
    应为 true(1);如果用其他方法,则 polish 的输入值应为 false(0)。

{
    void laguer(fcomplex a[], int m, fcomplex *x, int *its);
    int i,its,j,jj;
    fcomplex x,b,c,ad[MAXM];

    for (j=0;j<=m;j++) ad[j]=a[j];          对系数进行复制,以备逐次降价用
    for (j=m;j>=1;j--) {                    对每一个欲求的根循环:
        x=Complex(0.0,0.0);                 从零点开始以利于收敛到其余最小的一个根,
        laguer(ad,j,&x,&its);                  并找这个根
        if (fabs(x.i) <= 2.0*EPS*fabs(x.r)) x.i=0.0;
        roots[j]=x;
        b=ad[j];                            向前降价
        for (jj=j-1;jj>=0;jj--) {
            c=ad[jj];
            ad[jj]=b;
            b=Cadd(Cmul(x,b),c);
        }
    }
    if (polish)
        for (j=1;j<=m;j++)                  用原来未降价的系数修正各根
            laguer(a,m,&roots[j],&its);
    for (j=2;j<=m;j++) {                    通过直接插入法对根的实部排序
        x=roots[j];
        for (i=j-1;i>=1;i--) {
            if (roots[i].r <= x.r) break;
            roots[i+1]=roots[i];
        }
        roots[i+1]=x;
    }
}
```

### 9.5.4 本征值方法

一个矩阵 **A** 的本征值是"特征多项式"$P(x)=\det[\mathbf{A}-x\mathbf{1}]$的根,但正如我们在第11章中将要看到的,通过求根方法求本征值,一般来讲并不是一种有效的手段。换一个角度来考虑,我们可用本征值方法(第十一章中讨论的方法)对任意多项式的根进行修正。极易证明(参见

[6]),特殊 $m\times m$ 阶伴随矩阵

$$\mathbf{A}=\begin{pmatrix} -\frac{a_{m-1}}{a_m} & -\frac{a_{m-2}}{a_m} & \cdots & -\frac{a_1}{a_m} & -\frac{a_0}{a_m} \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & & & & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix} \tag{9.5.12}$$

的特征多项式与一般形式的多项式

$$P(x)=\sum_{i=0}^{m}a_ix^i \tag{9.5.13}$$

完全等价。如果系数 $a_i$ 是实数而不是复数,则 **A** 的本征值可以通过第11.5~11.6节中的子程序 **balanc** 和 **hqr** 求得(详见第十一章)。尽管本征值法的程序实现(**zrhqr** 见下)通常比 **zroots**(见上)慢 2 倍,但对某些类型的多项式来说,本征值法更能解决问题,这主要是因为 **hqr** 中的收敛算法太繁琐。在多项式系数为实数的情况下,如果用 **zroots** 求根不太顺利,我们建议读者不妨试试下面的 **zrhqr**。

```
#include "nrutil.h"
#define MAXIT 50

void zrhqr(float a[], int m, float rtr[], float rti[])
```

在给定实系数多项 $\sum_{i=0}^{m}a(i)x^i$ 的阶 m 及系数 a[0..m]的条件下,求解它的所有根。采用的方法是先构造一个较高阶的海森伯格(Hessenberg)矩阵(其本征值即为所求根),然后再利用子程序 balanc 和 hqr。根的实部和虚部将分别以 rtr[1..m]和 rti[1..m]返回。

```
{
    void balanc(float **a, int n);
    void hqr(float **a, int n, float wr[], float wi[]);
    int j,k;
    float **hess,xr,xi;

    hess=matrix(1,MAXM,1,MAXM);
    if (m > MAXM || a[m] == 0.0) nrerror("bad args in zrhqr");
    for (k=1;k<=m;k++) {                构造矩阵
        hess[1][k] = -a[m-k]/a[m];
        for (j=2;j<=m;j++) hess[j][k]=0.0;
        if (k != m) hess[k+1][k]=1.0;
    }
    balanc(hess,m);                     求本征值
    hqr(hess,m,rtr,rti);
    for (j=2;j<=m;j++) {                用直接插入法对根按其内部进行排序
        xr=rtr[j];
        xi=rti[j];
        for (k=j-1;k>=1;k--) {
            if (rtr[k] <= xr) break;
            rtr[k+1]=rtr[k];
            rti[k+1]=rti[k];
        }
        rtr[k+1]=xr;
        rti[k+1]=xi;
    }
    free_matrix(hess,1,MAXM,1,MAXM);
}
```

## 9.5.5 其他几种比较可靠的求根方法

在多项式求根的黑箱方法中(如 IMSL 库[3]中的求根算法,**詹金斯-特罗布(Jenkins-**

Traub)算法已成为一种实用标准;但由于这种方法过于复杂,这里不做讨论。读者参阅文献[4]。

**莱默-舒尔(Lehmer-Schur)算法**属于一类求根方法,这类方法将一维的划界概念加以推广,即把根孤立在复平面内,也就是在一个圆心和半径都给定的圆内,有效地确定是否含有多项式的根,然后,再对于何处放置试验圆作出一系列判断,以便分门别类地求出多项式的全部根。请读者参考文献[1]中的论述。

### 9.5.6 有关根修正的一些技巧

一旦确定出根所在的区间范围,那么用牛顿-拉斐森方法求实数根是很行之有效的。如第5.3节所述,多项式及其导数值可以很快地同时计算出来。对一个阶数为 n,系数为 c[0]...c[n]的多项式,下面这段小程序是牛顿-拉斐森方法的一个循环。

```
p=c[n]*x+c[n-1];
p1=c[n];
for(i=n-2;i>=0;i--) {
    p1=p+p1*x;
    p=c[i]+p*x;
}
if (p1 == 0.0) nrerror("derivative should not vanish");
x -= p/p1;
```

修正了一个多项式的所有实根之后,就须对所有复根进行修正,这时既可以用直接方法,也可通过二次因子。

若将上面那段小程序改写为复数类型的数据,则牛顿-拉斐森法即为修正复根的直接方法。对实系数的多项式,在设置初始预测值(试验解)时,应注意使其离开实轴,否则迭代时将永远无法脱离开实轴,而且可能会在多项式的极大或极小值处射向无穷远。

对实多项式,修正复根(这时也可称为重实根)的方法是**贝尔斯托(Bairstow)方法**,这种方法要求寻找二次因子,其好处在于避免了所有的复数运算。具体地说,也就是找一个二次因子,使其包含两个根 $x=a\pm ib$,即

$$x^2-2ax+(a^2+b^2)=x^2+Bx-C \tag{9.5.14}$$

一般地,如果我们以一个二次因子去除某个多项式,则剩下的将是一个线性项

$$P(x)=(x^2-Bx-C)Q(x)+Rx+S \tag{9.5.15}$$

如果给定 $B$ 和 $C$,则由多项式除法(见第5.3节)容易求得 $R$ 及 $S$。我们可以视 $R$、$S$ 为 $B$ 和 $C$ 的可调整的函数,如果这个二次因子为零则 $R$ 和 $S$ 均为零。

在某个根的邻域内,因 $B$、$C$ 的微小变化而引起的 $R$、$S$ 的改变可以用一阶泰勒级数展开来近似,即

$$R(B+\delta B,C+\delta C)\approx R(B,C)+\frac{\partial R}{\partial B}\delta B+\frac{\partial R}{\partial C}\delta C \tag{9.5.16}$$

$$S(B+\delta B,C+\delta C)\approx S(B,C)+\frac{\partial S}{\partial B}\delta B+\frac{\partial S}{\partial C}\delta C \tag{9.5.17}$$

为了计算偏导数,考虑式(9.5.15)对 $C$ 的导数;因为 $P(x)$ 为一固定的多项式,故与 $C$ 无关,从而有

$$0=(x^2+Bx+C)\frac{\partial Q}{\partial C}-Q(x)+\frac{\partial R}{\partial C}x+\frac{\partial S}{\partial C} \tag{9.5.18}$$

移项可得

$$-Q(x)=(x^2+Bx+C)\frac{\partial Q}{\partial C}+\frac{\partial R}{\partial C}x-\frac{\partial S}{\partial C} \tag{9.5.19}$$

同理，由于 $P(x)$ 与 $B$ 无关，因而将式(9.5.15)对 $B$ 作微分可以得到

$$-xQ(x) = (x^2 + Bx + C)\frac{\partial Q}{\partial B} + \frac{\partial R}{\partial B}x + \frac{\partial S}{\partial B} \tag{9.5.20}$$

比较式(9.5.19)和(9.5.15)可以发现，它们在形式上完全匹配。因此，如果对 $P(x)$ 作二次综合除法，即除以 $Q(x)$ 后得到 $R_1x+S_1$，则有

$$\frac{\partial R}{\partial C} = -R_1 \qquad \frac{\partial S}{\partial C} = -S_1 \tag{9.5.21}$$

为了求出偏导数的表达式，在二次因子的两个根 $x_+$ 和 $x_-$ 处计算式(9.5.20)的值。由于

$$Q(x_\pm) = R_1x_\pm + S_1 \tag{9.5.22}$$

我们有

$$\frac{\partial R}{\partial B}x_+ + \frac{\partial S}{\partial B} = -x_+(R_1x_+ + S_1) \tag{9.5.23}$$

$$\frac{\partial R}{\partial B}x_- + \frac{\partial S}{\partial B} = -x_-(R_1x_- + S_1) \tag{9.5.24}$$

从 $\frac{\partial R}{\partial B}$、$\frac{\partial S}{\partial B}$ 为未知量求解这两个方程，并注意到

$$x_+ + x_- = -B \qquad x_+x_- = C \tag{9.5.25}$$

可以得到

$$\frac{\partial R}{\partial B} = BR_1 - S_1 \qquad \frac{\partial S}{\partial B} = CR_1 \tag{9.5.26}$$

现在的贝尔斯托(Bairstow)方法包括在二维情况下，用牛顿-拉斐森算法(这正是下节中我们将要讨论的主要内容)，求使 $R$ 和 $S$ 同时为零的解。在每次循环中，需使用两次综合除法来算 $R$、$S$ 及其对 $B$、$C$ 的偏导数。和一维的牛顿-拉斐森方法一样，贝尔斯托方法在某一对根(或实或复)附近收敛效果不错，但若从任一随机点开始迭代则可能导致失败，因此我们建议读者只有在对试验复根进行修正时使用这一方法。

```
#include <math.h>
#include "nrutil.h"
#define ITMAX 20                        最大迭代次数
#define TINY 1.0e-6

void qroot(float p[], int n, float *b, float *c, float eps)
    给定一个 n 阶多项式的 n+1 个系数 p[0..n]以及一个二次因子 x*x+b*x-c 各系数的试验值，本程序将不断地
    修正问题的解，直到系数 b,c 的变化幅度小于 eps 为止，程序中调用了第5.3节中的 poldiv。
{
    void poldiv(float u[], int n, float v[], int nv, float q[], float r[]);
    int iter;
    float sc,sb,s,rc,rb,r,dv,delc,delb;
    float *q,*qq,*rem;
    float d[3];

    q=vector(0,n);
    qq=vector(0,n);
    rem=vector(0,n);
    d[2]=1.0;
    for (iter=1;iter<=ITMAX;iter++) {
        d[1]=(*b);
        d[0]=(*c);
        poldiv(p,n,d,2,q,rem);
        s=rem[0];                          第一次除法算 r,s.
        r=rem[1];
        poldiv(q,(n-1),d,2,qq,rem);
        sb = -(*c)*(rc = -rem[1]);          第二次除法求r,s对c的偏导数.
        rb = -(*b)*rc+(sc = -rem[0]);
        dv=1.0/(sb*rc-sc*rb);              解 2 × 2 的方程
        delb=(r*sc-s*rc)*dv;
```

```
        delc=(-r*sb+s*rb)*dv;
        *b += (delb=(r*sc-s*rc)*dv);
        *c += (delc=(-r*sb+s*rb)*dv);
        if ((fabs(delb) <= eps*fabs(*b) || fabs(*b) < TINY)
            && (fabs(delc) <= eps*fabs(*c) || fabs(*c) < TINY)) {
            free_vector(rem,0,n);          系数收敛
            free_vector(qq,0,n);
            free_vector(q,0,n);
            return;
        }
    }
    nrerror("Too many iterations in routine qroot");
}
```

前面曾经提到过，如果两个试探解经修正后收敛到一个根这将是非常令人恼火的，因为这时我们并不知道在修正过程中是不是丢失了一个根，也不知道是否真有一重根，而这个重根在前面的降阶步骤中仅仅由于舍入误差才被区分开。有一种解决办法是降阶并重新修正，但在修正阶段，需要避免的恰恰是降阶。这时，可以转而使用马赫里(Maehly)方法。马赫里指出，对降了阶的多项式

$$P_j(x) \equiv \frac{P(x)}{(x-x_1)\dots(x-x_j)} \tag{9.5.27}$$

其导数可以写成

$$P_j'(x) = \frac{P'(x)}{(x-x_1)\dots(x-x_j)} - \frac{P(x)}{(x-x_1)\dots(x-x_j)}\sum_{i=1}^{j}(x-x_i)^{-1} \tag{9.5.28}$$

因而牛顿—拉斐森方法中，由估计值 $x_k$ 推导下一估计值 $x_{k+1}$ 的公式可写为

$$x_{k+1} = x_k - \frac{P(x_k)}{P'(x_k) - P(x_k)\sum_{i=1}^{j}(x_k - x_i)^{-1}} \tag{9.5.29}$$

使用上述方程时，如果以 $i$ 标记已修正过的根，则可以避免某个试验解收敛到另一试验解所对应的真根上，这是一个以所谓消零替代真降阶的例子。

此外，前面提到过的米勒方法也是一种值得使用的根修正法。

参考文献和进一步读物：


Acton, F. S. 1970, *Numerical Methods That Work*, 1990, corrected editiion (Washington: Mathematical Association of America), Chapter 7. [1]

Peters G., and Wilkinson, J. H. 1971, *Journal of the Institute of Mathematics and its Applications*, vol. 8, pp. 16～35. [2]

*IMSL Math/Library Users Manual* (IMSL Inc., 2500 City West Boulevard, Houston TX 77042). [3]

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §8.9～8.13. [4]

Adams, D. A. 1967, *Communications of the ACM*, vol. 10, pp. 655～658. [5]

Johnson, L. W., and Riess, R. D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §4.4.3. [6]

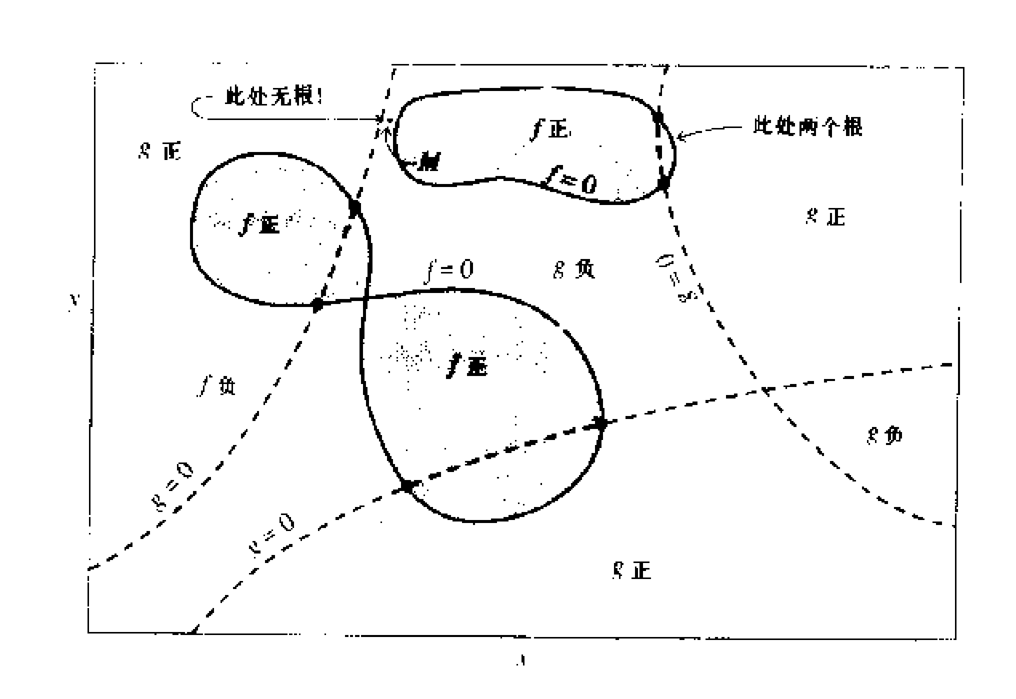

## 9.6 非线性方程系统的牛顿-拉斐森方法

对于含一个以上非线性方程的系统，目前还没有一种好的一般性解法。这一断言虽然极

端但却很难驳倒，而且我们不难理解为什么将来也(很可能)不会有好的通用算法。考虑二维情况，即求同时满足

$$
\begin{aligned}
f(x,y) &= 0 \\
g(x,y) &= 0
\end{aligned}
\tag{9.6.1}
$$

的解，其中 $f$ 和 $g$ 是两个任意函数，每个函数都具有一些零等高线，这些零等高线把 $(x,y)$ 平面分成若干区域，在这些区域内每个函数分别相应地取正值或负值。这里我们所感兴趣的仅仅是这些零等高线的边界，因为我们所要求的解(如果有的话)就是 $f$ 和 $g$ 之零等高线的交点(见图9.6.1)。但是一般情况下，$f$ 和 $g$ 相互之间根本没有任何联系！无论从 $f$ 的角度还是从 $g$ 的角度来分析，它们的公共点都没有什么特殊性质。为了找到所有的公共点，即求非线性方程组的解，(一般)需给出两个函数的全部零等高线。而且由于零等高线(一般)是由一些数目不定的不相交的闭曲线组成，那么，我们怎样才能知道全部闭曲线已经被找出来了呢？




其中实线代表 $f(x,y)$，虚线为 $g(x,y)$，每个方程将 $(x,y)$ 平面分成若干个正的和负的区域，这些区域的边界即为零等高线，我们所要求的解即为这些不相关的曲线的交点，但解的个数事前是未知的。

图9.6.1 图示为两个二元非线性方程的解

对于两维以上的问题，需要找到同时属于 $N$ 个不相关的零等值超平面的点，其中每个超平面的维数是 $N-1$ 维。如果没有相当出色的洞察力的话，求根简直是不可能的事。因为我们几乎总是需要利用一些附加的关于特定问题的特定信息，才能搞清楚诸如，"该问题是不是能得到一个唯一解"，以及"解大约在什么位置"之类的疑问。阿克(Acton)在他的书中[1]中，对这些特殊技巧作了一些尝试的有益的探讨。

本节中我们将介绍多维求根方法中最简单的一种，即牛顿-拉斐森方法，该方法对于足够好的初值，收敛效果很明显；在不收敛的情况下也能指出(虽然不是证明)假想的根不存

在。在第9.7节中我们将讨论复杂一些的牛顿-拉斐森法,它有望对其全局收敛性质作一番改进;第9.7节中,还将介绍一种称为布罗依顿(Broyden)的方法,它是对弦截法的多维推广。

一般地,给定含有 $N$ 个变量 $x_i(i=1,2,\ldots,N)$ 的 $N$ 个函数方程:

$$F_i(x_1,x_2,\ldots x_N)=0 \qquad i=1,2,\ldots,N \tag{9.6.2}$$

记 $\mathbf{x}$ 为以 $x_i$ 为分量的向量,$\mathbf{F}$ 为以 $F_i$ 为分量的向量,那么在 $\mathbf{x}$ 的领域内,每个函数 $F_i$ 可用泰勒级数展开为

$$F_i(\mathbf{x}+\delta\mathbf{x})=F_i(\mathbf{x})+\sum_{j=1}^{N}\frac{\partial F_i}{\partial x_j}\delta x_j+O(\delta\mathbf{x}^2) \tag{9.6.3}$$

出现在式(9.6.3)中的偏导数构成的矩阵是雅可比(Jacobian)矩阵 $\mathbf{J}$:

$$J_{ij}=\frac{\partial F_i}{\partial x_j} \tag{9.6.4}$$

式(9.6.3)以矩阵形式表示,即为

$$\mathbf{F}(\mathbf{x}+\delta\mathbf{x})=\mathbf{F}(\mathbf{x})+\mathbf{J}\cdot\delta\mathbf{x}+O(\delta\mathbf{x}^2) \tag{9.6.5}$$

忽略 $\delta\mathbf{x}^2$ 项及其更高阶项,并置 $\mathbf{F}(\mathbf{x}+\delta\mathbf{x})=0$,可以得到一个关于修正项 $\delta\mathbf{x}$ 的线性方程组,即

$$\mathbf{J}\cdot\delta\mathbf{x}=-\mathbf{F} \tag{9.6.6}$$

其中 $\delta\mathbf{x}$ 可使每个函数都同时接近于零。

对式(9.6.6)所示的矩阵方程,用第2.3节的 $LU$ 分解方法来求解,并将求出的修正项添加到解向量中

$$\mathbf{x}_{new}=\mathbf{x}_{old}+\delta\mathbf{x} \tag{9.6.7}$$

这一过程不断迭代直到收敛为止。一般情况下,最好检查一下函数和变量已收敛的程度,当其中一个达到机器精度时,另一个就不会改变了。

下面这段程序(**mnewt**)将从初始解向量 x[1..n]开始,执行 ntrial 次迭代。当函数 $F_i$ 的大小之和小于某一容差限 tolf,或者修正项 $\delta x_i$ 的绝对值之和小于容差限 tolx 时,迭代过程将终止。**mnewt** 中调用了一个由用户定义的函数 **usrfun**,它提供函数值 **F** 及雅可比矩阵 **J**。若 **J** 的解析计算太复杂,不妨在 **usrfun** 中调用第9.7节中的 **fdjac**,利用有限差分来计算偏导数,还应注意迭代次数 ntrial 不能取的太大,而且在进入下一轮迭代以前,最好观察一下当前的运行结果。

```
#include <math.h>
#include "nrutil.h"

void usrfun(float *x, int n, float *fvec, float **fjac);
#define FREERETURN {free_matrix(fjac,1,n,1,n);free_vector(fvec,1,n);\
    free_vector(p,1,n);free_ivector(indx,1,n);return;}

void mnewt(int ntrial, float x[], int n, float tolx, float tolf)
```

给定一个根的初值 x[1..n],通过 ntrial 步牛顿-拉斐森法修正这个根。当各变量的绝对增量之和小于 tolx 或函数的绝对值之和小于 tolf 时,迭代终止。

```
{
    void lubksb(float **a, int n, int *indx, float b[]);
    void ludcmp(float **a, int n, int *indx, float *d);
    int k,i,*indx;
    float errx,errf,d,*fvec,**fjac,*p;
```

```
    indx=ivector(1,n);
    p=vector(1,n);
    fvec=vector(1,n);
    fjac=matrix(1,n,1,n);
    for (k=1;k<=ntrial;k++) {
        usrfun(x,n,fvec,fjac);          用户定义函数，fvec表示函数在x处的值，fjac代
        errf=0.0;                           表雅可比矩阵
        for (i=1;i<=n;i++) errf += fabs(fvec[i]);     检查函数收敛
        if (errf <= tolf) FREERETURN
        for (i=1;i<=n;i++) p[i] = -fvec[i];       线性方程的右端项
        ludcmp(fjac,n,indx,&d);           用LU分解法解线性方程
        lubksb(fjac,n,indx,p);
        errx=0.0;                         检查根收敛
        for (i=1;i<=n;i++) {              更新解
            errx += fabs(p[i]);
            x[i] += p[i];
        }
        if (errx <= tolx) FREERETURN
    }
    FREERETURN
}
```

### 9.6.1 牛顿法与极小化

在下一章中，我们将会看到，对于多元函数求极小值的问题，目前已有一些很有效的一般性技巧。读者也许会问：为什么这类问题往往（相对）比较容易，而多维情况下的求根常常很难？极小化问题难道不是等价于求使 $N$ 维梯度向量为零的点，也就是说与 $N$ 元函数求根没有多大区别吗？不是的，因为一个梯度向量的各个分量不是独立的、任意的函数，相反，它们服从一些极其严格的所谓可积条件。粗略地说，沿一个单独平面下降，总能找到一个极小值，因此，"下降"检验是一维的，而对多维求根则没有一种类似的概念上的方法可寻。在这种情况下，"下降"意味着从 $N$ 个独立的函数空间同时下降，因此，决定某一个维上要进展多少值需与另一维的进展情况作比较，这样在这一问题上必然允许有大量的折衷方案。

对多维求根问题，读者可能会想到把所有维数拆成一维，即，把每个函数 $F_i$ 的平方和相加，得一个主函数 $F$。这个主函数满足(i)具有正定性；(ii)在原非线性方程组的所有解处恰好有一个全局极小值零点。但是，正如我们在下一章中将要看到的，那些求极小值的有效算法对全局和局部极小值根本不加区分，而且非常令人失望的是，函数 F 常常有很多个局部极小值。例如在图9.6.1中，只要 $f$ 和 $g$ 的零等高线靠得很近，就有可能存在一个局部极小值。图中标号为 $M$ 的点即是如此，但可以看出，在 $M$ 点附近并没有根。

不管怎样，多维求根实际上还是可以利用极小化的思想的，方式是将主函数 $F$ 与牛顿法相结合，并将牛顿法应用于函数 $F_i$ 的全体集合。尽管这种方法有时只能收敛到 $F$ 的某个局部极小点，但对单独用牛顿法不能解决的问题一般它都能求解。下节中我们将详细讨论这些方法。

参考文献和进一步读物：


Acton, F. S. 1970, *Numerical Methods That Work*, 1990, corrected edition (Washington: Mathematical Association of America), Chapter 14. [1]

## 9.7 非线性方程系统的全局收敛法

我们已经看到在求解非线方程时，如果初始预测值没有足够地接近根，则牛顿法有一个不幸的趋势，会偏离到无规则的远处。而**全局法**对于几何任意初始值都能收敛到解。本节我们将发展一种算法，它是将快速局部收敛的牛顿法和一个全局收敛策略结合起来，这个全局收敛策略在每一步迭代中，将保证某些进程趋于解。这个算法和第10.7节中将要详细讨论的拟牛顿法的极小化是密切相关的。

回顾第9.6节中的讨论，对于方程组的牛顿法步骤是

$$\mathbf{F}(\mathbf{x}) = 0 \tag{9.7.1}$$

令

$$\mathbf{x}_{new} = \mathbf{x}_{old} + \delta\mathbf{x} \tag{9.7.2}$$

其中

$$\delta\mathbf{x} = -\mathbf{J}^{-1} \cdot \mathbf{F} \tag{9.7.3}$$

$\mathbf{J}$ 是雅可比矩阵。我们怎样决定是否接收牛顿步长 $\delta\mathbf{x}$ 呢？一个合理的策略是，要求这个步长使 $|\mathbf{F}|^2 = \mathbf{F} \cdot \mathbf{F}$ 下降。这就相同于我们强加的要求，使下式 $f$ 极小化

$$f = \frac{1}{2}\mathbf{F} \cdot \mathbf{F} \tag{9.7.4}$$

（为了后面的方便才有这$\frac{1}{2}$）。每次求解式(9.7.1)，同时极小化式(9.7.4)，但是可能式(9.7.4)的局部最小值使式(9.7.1)无解。因此，如前所述，简单地从第十章找一个求极小值算法运用于式(9.7.4)，这不是一种很好的思想。

为了推出一种较好的策略，注意到牛顿法步长式(9.7.3)对 $f$ 是下降方向：

$$\nabla f \cdot \delta\mathbf{x} = (\mathbf{F} \cdot \mathbf{J}) \cdot (-\mathbf{J}^{-1} \cdot \mathbf{F}) = -\mathbf{F} \cdot \mathbf{F} < 0 \tag{9.7.5}$$

所以我们的策略是非常简单：首先，我们常常试着采用牛顿全步长，因为一旦我们是足以逼近所求解，那么我们将得到二次收敛。然而，在每一次迭代过程中，我们要检查这个所建议的步长是否减少 $f$ 值。若不，我们就应回过头来沿牛顿方向追踪，直到我们有一个可接受的步长为止。牛顿步长对 $f$ 而言是下降方向，以保证我们用回溯方法可以找到一个可接受的步长。下面我们将详细讨论这种回溯算法。

注意这种方法本质上是，采用使 F 等于零所设计的牛顿步长来求 $f$ 极小化，而不是等价于，采用使 $\nabla f$ 等于零所设计的牛顿步长来直接求 $f$ 极小化。当着落在 $f$ 的局部极小值时，这方法仍然可能发生失败，但这种情况在实践中是非常罕见的。在下面的程序 **newt** 中，若这种情况发生它就会告诫用户。而补救的方法是用一个新的初始点重新试验。

### 9.7.1 线性搜索和回溯

当我们没有足够接近 $f$ 的极小值时，就无需取全牛顿步长 $\mathbf{p} = \delta\mathbf{x}$ 来下降函数，为了使二次收敛有效，我们可以移动很远。当我们沿着牛顿方向移动，我们已保证了 $f$ 一开始就是下降的。所以，移动的目标是沿牛顿步长 $\mathbf{p}$ 的方向移动到一个新的点 $\mathbf{x}_{new}$，但不需要移动步长的全部路程：

$$\mathbf{x}_{new} = \mathbf{x}_{old} + \lambda\mathbf{p} \qquad 0 < \lambda < 1 \tag{9.7.6}$$

目的是要寻找 $\lambda$ 值以使 $f(\mathbf{x}_{old} + \lambda\mathbf{p})$ 充分下降。早在1970年以前，标准的方法是，选择 $\lambda$ 以使 $\mathbf{x}_{new}$ 在 $\mathbf{p}$ 方向上

确实使 $f$ 达到极小化。但是，现在我们知道这样做就要计算函数值，这是极大的浪费。一个比较好的策略是：因为在我们的算法中，$\mathbf{p}$ 方向常常是牛顿方向，所以我们首先选 $\lambda=1$，全牛顿步长。当 $\mathbf{x}$ 是充分接近解时，这就将导致二次收敛。但是，若 $f(\mathbf{x}_{new})$ 不能满足我们可接受的准则，则我们沿着牛顿方向回溯，试着使用较小的 $\lambda$ 值，直到我们找到一个合适点。因为牛顿方向是下降方向，所以保证了对于足够小的 $\lambda$ 可使 $f$ 递减。

对于可接受的步长采用什么准则呢？若只要求 $f(\mathbf{x}_{new})<f(\mathbf{x}_{old})$ 是不充分的。这个准则在下面二个情况之一中，会使收敛到 $f$ 的最小值发生失败。第一种情况，构造的满足准则的步长序列其使 $f$ 下降速度相对于步长的长度太慢。第二种，有一组步长序列，其步长长度相对 $f$ 下降的初始速率太小。(这些序列的例子见[1]，117页)。

处理第一种问题的简单方法是，要求 $f$ 下降的平均速率至少是 $\nabla f\cdot\mathbf{p}$ 初始下降率的 $\alpha$ 部分：

$$f(\mathbf{x}_{new})\leq f(\mathbf{x}_{old})-\alpha\nabla f\cdot(\mathbf{x}_{new}-\mathbf{x}_{old}) \tag{9.7.7}$$

其中参数 $\alpha$ 满足 $0<\alpha<1$。我们能够避免得到非常小的 $\alpha$ 值，好的选择是 $\alpha=10^{-4}$。

第二种问题可以用这种方法来固定，即要求在 $\mathbf{x}_{new}$ 处 $f$ 下降率大于在 $\mathbf{x}_{old}$ 处 $f$ 下降率的 $\beta$ 部分，实践中，我们常常不需要改进这个第二种约束，因为我们的回溯算法将有一个内在中止，以避免步长取得太小。

对于实际的回溯算法程序有一策略：

定义

$$g(\lambda)\equiv f(\mathbf{x}_{old}+\lambda\,\mathbf{p}) \tag{9.7.8}$$

以使

$$g'(\lambda)=\nabla f\cdot\mathbf{p} \tag{9.7.9}$$

若我们需要回溯的话，则我们用已有的最近信息构造模型 $g$，并且选择 $\lambda$ 使这模型极小化。我们可以用有效的 $g(0)$ 和 $g'(0)$ 开始。通常，第一步是牛顿长，$\lambda=1$。若这一步没有接收，我们有了有效的 $g(1)$，所以我们构造模型 $g(\lambda)$ 为二次型：

$$g(\lambda)\approx[g(1)-g(0)-g'(0)]\lambda^2+g'(0)\lambda+g(0) \tag{9.7.10}$$

求这二次型的微分，当

$$\lambda=\frac{-g'(0)}{2[g(1)-g(0)-g'(0)]} \tag{9.7.11}$$

二次型取极小。因为牛顿步长失败，所以我们能够证明对于小的 $\alpha$，$\lambda\lesssim\frac{1}{2}$。但是，我们需要警惕防止 $\lambda$ 太小的值，一般设 $\lambda_{min}=0.1$。

在第二次和以后的回溯过程中，我们取模型 $g$ 为 $\lambda$ 的三次型，利用原先 $g(\lambda_1)$ 值和当前第二个 $g(\lambda_2)$ 的值来构造：

$$g(\lambda)=a\lambda^3+b\lambda^2+g'(0)\lambda+g(0) \tag{9.7.12}$$

在 $\lambda_1$ 和 $\lambda_2$ 处给出上述表达式 $g$ 的当前值，构成二个方程，由此求出系数 $a$ 和 $b$：

$$\begin{bmatrix}a\\b\end{bmatrix}=\frac{1}{\lambda_1-\lambda_2}\begin{bmatrix}1/\lambda_1^2 & -1/\lambda_2^2\\ -\lambda_2/\lambda_1^2 & \lambda_1/\lambda_2^2\end{bmatrix}\cdot\begin{bmatrix}g(\lambda_1)-g'(0)\lambda_1-g(0)\\ g(\lambda_2)-g'(0)\lambda_2-g(0)\end{bmatrix} \tag{9.7.13}$$

(9.7.12)式三次型的极小值是在

$$\lambda=\frac{-b+\sqrt{b^2-3ag'(0)}}{3a} \tag{9.7.14}$$

我们强制使 $\lambda$ 处于 $\lambda_{max}=0.5\lambda_1$ 和 $\lambda_{min}=0.2\lambda_1$ 之间。

程序有二个附加特性值，最小步长长度 alamin 和最大步长长度 stpmax。**lnsrch** 也可用于下节拟牛顿极小化程序 **dfpmin** 中。

```
#include <math.h>
#include "nrutil.h"
#define ALF 1.0e-4                 保证函数值充分下降
```

```
#define TOLX 1.0e-7                    Δx的收敛准则

void lnsrch(int n, float xold[], float fold, float g[], float p[], float x[], float *f,
    float stpmax, int *check, float (*func)(float []))
```

给定一个n维的点xold[1..n]，函数值fold和斜率g[1..n]以及方向p[1..n]，从xold沿着方向p寻找一个新的点x[1..n]，那里使函数func"充分地"下降。新的函数返回到f。stpmax是限制步长长度的输入量，以使在函数没有定义的范围或者溢出的情况下都不计算函数。p通常是牛顿方向。输出量check是假(0)：正常退出；是真(1)，表示x是非常接近xold。在极小化算法中，这通常是单调收敛的和可被忽略。但是，在零点寻找算法中，正在调用的程序应该检查收敛是否是伪的。在本程序中，某些"难"题可以要求双精度。

```
{
    int i;
    float a,alam,alam2,alamin,b,disc,f2,fold2,rhs1,rhs2,slope,sum,temp,
        test,tmplam;

    *check=0;
    for (sum=0.0,i=1;i<=n;i++) sum += p[i]*p[i];
    sum=sqrt(sum);
    if (sum > stpmax)
        for (i=1;i<=n;i++) p[i] *= stpmax/sum;      测量所尝试的步长是否太长
    for (slope=0.0,i=1;i<=n;i++)
        slope += g[i]*p[i];
    test=0.0;                                          计算λmin。
    for (i=1;i<=n;i++) {
        temp=fabs(p[i])/FMAX(fabs(xold[i]),1.0);
        if (temp > test) test=temp;
    }
    alamin=TOLX/test;
    alam=1.0;                                      通常首先尝试全牛顿步长
    for (;;) {                                     迭代循环开始
        for (i=1;i<=n;i++) x[i]=xold[i]+alam*p[i];
        *f=(*func)(x);
        if (alam < alamin) {                       在Δx上收敛。对于零寻找法，所调用
            for (i=1;i<=n;i++) x[i]=xold[i];          的程序将证明这收敛性
            *check=1;
            return;
        } else if (*f <= fold+ALF*alam*slope) return;      函数充分下降
        else {                                             回溯
            if (alam == 1.0)
                tmplam = -slope/(2.0*(*f-fold-slope));     第一次
            else {                                         继续回溯
                rhs1 = *f-fold-alam*slope;
                rhs2=f2-fold2-alam2*slope;
                a=(rhs1/(alam*alam)-rhs2/(alam2*alam2))/(alam-alam2);
                b=(-alam2*rhs1/(alam*alam)+alam*rhs2/(alam2*alam2))/(alam-alam2);
                if (a == 0.0) tmplam = -slope/(2.0*b);
                else {
                    disc=b*b-3.0*a*slope;
                    if (disc<0.0) nrerror("Roundoff problem in lnsrch.");
                    else tmplam=(-b+sqrt(disc))/(3.0*a);
                }
                if (tmplam>0.5*alam)
                    tmplam=0.5*alam;                λ ≤ 0.5λ1。
            }
        }
        alam2=alam;
        f2 = *f;
        fold2=fold;
        alam=FMAX(tmplam,0.1*alam);                 λ ≥ 0.1λ1。
    }                                               重新再试
}
```

现在，下面是全局收敛牛顿法程序 **newt**，它用程序 **lnsrch**。程序 **newt** 的特点是，用户不需提供解析的

雅可比矩阵；这程序将使用程序 **fdjac** 中的有限差分，试着来计算必须的 **F** 的偏导。这程序为了计算数值导数，使用了第5.7节中讨论的某些技巧。当然，如果对用户来说是容易的话，通常就能用一个数值计算雅可比行列式的程序来代替 **fdjac**。

```
#include <math.h>
#include "nrutil.h"
#define MAXTTS 200
#define TOLF 1.0e-4
#define TOLMIN 1.0e-6
#define TOLX 1.0e-7
#define STPMX 100.0
```

其中 MAXITS 是迭代的最大次数；TOLF 设置了函数值收敛的判别准则；TOLMIN 设置了判决是否有假收敛于极小值 fmin 发生的判别准则；TOLX 是对 $\delta x$ 收敛的判别准则；STPMX 是所允许的在线性搜索中所测量的最大步长长度。

```
int nn;                          与 fmin 用全局变量进行通信
float *fvec;
void (*nrfuncv)(int n, float v[], float f[]);
#define FREERETURN {free_vector(fvec,1,n);free_vector(xold,1,n);\
    free_vector(p,1,n);free_vector(g,1,n);free_matrix(fjac,1,n,1,n);\
    free_ivector(indx,1,n);return;}

void newt(float x[], int n, int *check,
    void(*vecfunc)(int, float [], flaot []))
```

给定一个 n 维根的初始猜测值 x[1..n]，用全局收敛牛顿法找根。使用用户提供的程序 **vecfunc**(n,x,fvec)返回函数等于零的向量，在程序中称为 fvec[1..n]。输出量 check 为假(0)是正常返回；若收敛到一个函数的局部极小值，定义为 fmin，则 check 为真(1)。在这种情况下，试着从一个不同的初始猜测值重新开始。

```
{
    void fdjac(int n, float x[], float fvec[], float **df,
        void (*vecfunc)(int, float [], float []));
    float fmin(float x[]);
    void lnsrch(int n, float xold[], float fold, float g[], float p[], float x[],
         float *f, float stpmax, int *check, float (*func)(float []));
    void lubksb(float **a, int n, int *indx, float b[]);
    void ludcmp(float **a, int n, int *indx, float *d);
    int i,its,j,*indx;
    float d,den,f,fold,stpmax,sum,temp,test,**fjac,*g,*p,*xold;

    indx=ivector(1,n);
    fjac=matrix(1,n,1,n);
    g=vector(1,n);
    p=vector(1,n);
    xold=vector(1,n);
    fvec=vector(1,n);                          定义全局变量
    nn=n;
    nrfuncv=vecfunc;
    f=fmin(x);                                 由此调用计算 fvec
    test=0.0;                                  测试初始猜测根用比TOLF更直接的测试
    for (i=1;i<=n;i++)
        if (fabs(fvec[i]) > test) test=fabs(fvec[i]);
    if (test<0.01*TOLF) FREERETURN
    for (sum=0.0,i=1;i<=n;i++) sum += SQR(x[i]);    为线性搜索计算 stpmax
    stpmax=STPMX*FMAX(sqrt(sum),(float)n);
    for (its=1;its<=MAXITS;its++) {            迭代循环开始
        fdjac(n,x,fvec,fjac,vecfunc);
        若解析的雅可比行列式是有效的, 用户可以用自己程序替换程序 fdjac
        own routine.
        for (i=1;i<=n;i++) {                   对线性搜索计算∇f
            for (sum=0.0,j=1;j<=n;j++) sum += fjac[j][i]*fvec[j];
            g[i]=sum;
```

```
    }
    for (i=1;i<=n;i++) xold[i]=x[i];        存储 x,
    fold=f;                                  和 f.
    for (i=1;i<=n;i++) p[i] = -fvec[i];    线性方端组的右端
    ludcmp(fjac,n,indx,&d);                用LU分解法求解线性方程
    lubksb(fjac,n,indx,p);
    lnsrch(n,xold,fold,g,p,x,&f,stpmax,check,fmin);
    返回新的x和f。当调用fmin时在新的x处也计算fvec
    test=0.0;                              为函数值收敛而测试
    for (i=1;i<=n;i++)
        if (fabs(fvec[i]) > test) test=fabs(fvec[i]);
    if (test < TOLF) {
        *check=0;
        FREERETURN
    }
    if (*check) {                          检测f的梯度为零,即伪收敛
        test=0.0;
        den=FMAX(f,0.5*n);
        for (i=1;i<=n;i++) {
            temp=fabs(g[i])*FMAX(fabs(x[i]),1.0)/den;
            if (temp > test) test=temp;
        }
            *check=(test < TOLMIN ? 1 : 0);
            FREERETURN
        }
        test=0.0;                              为δx收敛而测试
        for (i=1;i<=n;i++) {
            temp=(fabs(x[i]-xold[i]))/FMAX(fabs(x[i]),1.0);
            if (temp > test) test=temp;
        }
        if (test < TOLX) FREERETURN
    }
    nrerror("MAXITS exceeded in newt");
}
```

```
#include <math.h>
#include "nrutil.h"
#define EPS 1.0e-4                        逼近机器精度的平方根

void fdjac(int n, float x[], float fvec[], float **df, void(*vecfunc)(int, float [], float []))
```

计算正向差分以逼近雅可比行列式。输入 x[1..n]是雅可比行列式被计算的点,fvec[1..n]是在那点的函数值向量,以及 vecfunc(n,x,f)是一个用户所提供的程序,它返回 x 点处的函数向量,输出 df[1..n][1..n]是雅可比矩阵。

```
{
    int i,j;
    float h,temp,*f;

    f=vector(1,n);
    for (j=1;j<=n;j++) {
        temp=x[j];
        h=EPS*fabs(temp);
        if (h == 0.0) h=EPS;
        x[j]=temp+h;                                  减少有限精度误差的窍门
        h=x[j]-temp;
        (*vecfunc)(n,x,f);
        x[j]=temp;
        for(i=1;i<=n;i++) df[i][j]=(f[i]-fvec[i])/h;   正向差分公式
    }
    free_vector(f,1,n);
}
```

```
#include "nrutil.h"
```

```
extern int nn;
extern float *fvec;
extern void (*nrfuncv)(int n, float v[], float f[]);

float fmin(float x[])
返回 x 处的 f=1/2F·F。全局指针 *nrfuncv 指向那个返回 x 处函数向量的程序。在调用程序中它设置为指向用户
所提供的程序。全局变量也传输函数值返回到所调用的程序中去。
{
    int i;
    float sum;

    (*nrfuncv)(nn,x,fvec);
    for (sum=0.0,i=1;i<=nn;i++) sum += SQR(fvec[i]);
    return 0.5*sum;
}
```

程序 **newt** 假设 **x** 和 **F** 的所有分量的典型值是顺序单位，若这假设被极坏地违反了，程序就会失败，若这种情况发生，在调用 **newt** 以前，应该用它们的典型值重新换算这些变量。

## 9.7.2 多维弦截法：布罗依登法

上述实施的牛顿法是功能十分强大的，但它仍然有几个缺点。它需要有雅可比行列式矩阵。在许多问题中，解析的导数也是得不到的。若函数的估算非常费时，则由于雅可比行列式的有限差分测定的代价使这方法的使用受到了禁止。

正如在第10.7节中讨论的拟牛顿法，它提供了在极小化算法中，海赛(Hessian)矩阵的容易计算的近似值。拟牛顿法也提供了在寻找零值中，雅可比矩阵的容易计算的近似值。这些方法通常也称为**弦截法**，因为在一维情况下，它们退化为弦截法(第9.2节)(参阅文献[1])。下面首先介绍的这种方法是这些方法中最好的一种，即布罗依登法(Broyden's method)[2]。

我们用 **B** 表示近似的雅可比矩阵。那么，第 i 个拟牛顿步长 $\delta\mathbf{x}_i$ 是下列方程的解

$$\mathbf{B}_i \cdot \delta\mathbf{x}_i = -\mathbf{F}_i \tag{9.7.15}$$

其中 $\delta\mathbf{x}_i=\mathbf{x}_{i+1}-\mathbf{x}_i$(参见方程(9.7.3))。拟牛顿条件或弦截条件是 $\mathbf{B}_{i+1}$ 满足

$$\mathbf{B}_{i+1} \cdot \delta\mathbf{x}_i = \delta\mathbf{F}_i \tag{9.7.16}$$

其 $\delta\mathbf{F}_i=\mathbf{F}_{i+1}-\mathbf{F}_i$。这是一维弦截法逼近导数 $\delta F/\delta x$ 的推广。但是在大于一维情况下，式(9.7.16)不能唯一确定 $\mathbf{B}_{i+1}$。

许多约束 $\mathbf{B}_{i+1}$ 的辅助条件已经被研究，而在实践中最好的执行算法是由布罗依登的公式得到。这个公式是基于这样的思路，它是使 $\mathbf{B}_i$ 的最小变化与弦截方程式(9.7.16)一致从而得到 $\mathbf{B}_{i+1}$。布罗依登指出这个最终公式是

$$\mathbf{B}_{i+1} = \mathbf{B}_i + \frac{(\delta\mathbf{F}_i - \mathbf{B}_i \cdot \delta\mathbf{x}_i) \otimes \delta\mathbf{x}_i}{\delta\mathbf{x}_i \cdot \delta\mathbf{x}_i} \tag{9.7.17}$$

很容易检验 $\mathbf{B}_{i+1}$ 满足式(9.7.16)。

布罗依登方法的早期实现采用谢尔曼-莫里森公式，即式(2.7.2)，解析地求式(9.7.17)的逆，

$$\mathbf{B}_{i+1}^{-1} = \mathbf{B}_i^{-1} + \frac{(\delta\mathbf{x}_i - \mathbf{B}_i^{-1} \cdot \delta\mathbf{F}_i) \otimes \delta\mathbf{x}_i \cdot \mathbf{B}_i^{-1}}{\delta\mathbf{x}_i \cdot \mathbf{B}_i^{-1} \cdot \delta\mathbf{F}_i} \tag{9.7.18}$$

那么，人们就可通过 $O(N^2)$ 次矩阵乘法操作来确定

$$\delta\mathbf{x}_i = -\mathbf{B}_i^{-1} \cdot \mathbf{F}_i \tag{9.7.19}$$

由此替换用 $LU$ 分解方程(9.7.3)。这方法的缺点是它不能很容易地嵌入到全局收敛策略中去。因为在全局收敛策略中，求方程(9.7.4)式的梯度需要用 **B**，而不是 $\mathbf{B}^{-1}$，

$$\nabla(\frac{1}{2}\mathbf{F}\cdot\mathbf{F}) \simeq \mathbf{B}^T\cdot\mathbf{F} \tag{9.7.20}$$

相应地，在式(9.7.17)中我们实行更改的公式。

然而，我们仍然保留了 $O(N^2)$ 次操作求解式(9.7.5)，但是 $QR$ 分解(第2.10节)替换了 $LU$ 分解，其理由是因为(9.7.17)方程的特殊形式，通过 $O(N^2)$ 次操作 $\mathbf{B}_i$ 的 $QR$ 分解，可以被更新成 $\mathbf{B}_{i+1}$ 的 $QR$ 分解。我们所需的全部只是用一个初始近似值 $\mathbf{B}_0$ 去进行，通常起始时，用一个简单的单位矩阵是可行的，然后允许 $O(N)$ 次更新，以便产生一个合理的雅可比行列式的近似值。但是，我们更愿意通过调用 **fdjac**，使用有限差分近似方法，计算前第 $N$ 个函数值来初始化 $\mathbf{B}_0$。

因为 $\mathbf{B}$ 不是精确的雅可比行列式，所以我们不能保证 $\delta\mathbf{x}$ 对于 $f=\frac{1}{2}\mathbf{F}\cdot\mathbf{F}$(即式(9.7.5))是在下降的方向。因此，如果 $\mathbf{B}$ 偏离真实的雅可夫行列式很远，那么线性搜索算法不能返回一个合适步长。在这种情况下，我们需要另外调用 fdjac 重新初始化 $\mathbf{B}$。

和一维情况中的弦截法相同，一旦足以接近根值，布罗依登法将超线性地收敛。将这方法嵌入到全局策略中，它几乎也和牛顿法一样稳健，并且通常需要极少几个函数值的计算来确定零点。注意，甚至方法收敛的，$\mathbf{B}$ 的最后值常常也不是接近根处真正的雅可比行列式。

下面给出程序 **broydn** 在结构上非常类似于程序 **newt**，而基本差别是用 $QR$ 分解替换 $LU$ 分解，并且用更新的公式替换直接地确定雅可比行列式，在程序 **newt** 末尾关于变量测度的标记同样可用于 **broydn** 中。

```
#include <math.h>
#inclued "nrutil.h"
#define MAXITS 200
#define EPS 1.0e-7
#define TOLF 1.0e-4
#define TOLX EPS
#define STPMX 100.0
#define TOLMIN 1.0e-6
```

其中 MAXITS 是迭代的最大次数；EPS 是接近机器的精度数；TOLF 是函数值收敛的判别准则；TOLX 是 $\delta x$ 的收敛判别准则；STPMX 是在线性搜索中所允许的、测量的最大步长长度；TOLMIN 是用来判决是否有假收敛于极小值 $f_{min}$ 的事件发生。

```
#define FREERETURN {free_vector(fvec,1,n);free_vector(xold,1,n);\
    free_vector(w,1,n);free_vector(t,1,n);free_vector(s,1,n);\
    free_matrix(r,1,n,1,n);free_matrix(qt,1,n,1,n);free_vector(p,1,n);\
    free_vector(g,1,n);free_vector(fvcold,1,n);free_vector(d,1,n);\
    free_vector(c,1,n);return;}

int nn;                         与 fmin进行全局变量通信
float *fvec;
void (*nrfuncv)(int n, float v[], float f[]);

void broydn(float x[], int n, int *check, void (*vecfunc)(int, float [], float []))
```

给定一个 $n$ 维的根的初始猜测值，用布罗依登法嵌入全局收敛策略中的方法来寻找根。使用用户提供的程序 vecfunc(n,x,fvec)返回函数等于零的向量，在程序中称为向量 fvec[1..n]。程序 fdjac 和程序 newt 中的函数 fmin 都被使用。输出量 check 为假(0)是正常返回；若程序收敛到一个函数的局部极小值 fmin，或者布罗依登方法没有进一步进展，则 check 为真(1)。在这种情况下，试着从一个不同的初始猜测值重新开始。

```
{
    void fdjac(int n, float x[], float fvec[], float **df,
        void (*vecfunc)(int, float [], float []));
    float fmin(float x[]);
    void lnsrch(int n, float xold[], float fold, float g[], float p[], float x[],
        float *f, float stpmax, int *check, float (*func)(float []));
    void qrdcmp(float **a, int n, float *c, float *d, int *sing);
    void qrupdt(float **r, float **qt, int n, float u[], float v[]);
    void rsolv(float **a, int n, float d[], float b[]);
```

```
int i,its,j,k,restrt,sing,skip;
float den,f,fold,stpmax,sum,temp,test,*c,*d,*fvcold;
float *g,*p,**qt,**r,*s,*t,*w,*xold;

c=vector(1,n);
d=vector(1,n);
fvcold=vector(1,n);
g=vector(1,n);
p=vector(1,n);
qt=matrix(1,n,1,n);
r=matrix(1,n,1,n);
s=vector(1,n);
t=vector(1,n);
w=vector(1,n);
xold=vector(1,n);
fvec=vector(1,n);                          定义全部变量
nn=n;
nrfuncv=vecfunc;
f=fmin(x);                                 在这调用时也计算向量 fvec
test=0.0;
for (i=1;i<=n;i++)                         测试初始猜测是根吗? 采用TOLF更直接的
    if (fabs(fvec[i]) > test)test=fabs(fvec[i]);          测试
if (test<0.01*TOLF) FREERETURN
for (sum=0.0,i=1;i<=n;i++) sum += SQR(x[i]);     为线性搜索计算stpmax
stpmax=STPMX*FMAX(sqrt(sum),(float)n);
restrt=1;                                  保证初始雅可比行列式得到计算
for (its=1;its<=MAXITS;its++) {            迭代循环开始
    if (restrt) {
        fdjac(n,x,fvec,r,vecfunc);         在r中初始化或者重新初始化雅可比行列式
        qrdcmp(r,n,c,d,&sing);             雅可比行列式QR分解
        if (sing) nrerror("singular Jacobian in broydn");
        for (i=1;i<=n;i++) {                           从 Q^T 显见
            for (j=1;j<=n;j++) qt[i][j]=0.0;
            qt[i][i]=1.0;
        }
        for (k=1;k<n;k++) {
            if (c[k]) {
                for (j=1;j<=n;j++) {
                    sum=0.0;
                    for (i=k;i<=n;i++)
                        sum += r[i][k]*qt[i][j];
                    sum /= c[k];
                    for (i=k;i<=n;i++)
                        qt[i][j] -= sum*r[i][k];
                }
            }
        }
        for (i=1;i<=n;i++) {                           从 R 显见
            r[i][i]=d[i];
            for (j=1;j<i;j++) r[i][j]=0.0;
        }
    } else {                                           执行Broyden 更新
        for (i=1;i<=n;i++) s[i]=x[i]-xold[i];          s = δx.
        for (i=1;i<=n;i++) {                           t = R · s.
            for (sum=0.0,j=i;j<=n;j++) sum += r[i][j]*s[j];
            t[i]=sum;
        }
        skip=1;
        for (i=1;i<=n;i++) {                           w = δF − B · s.
            for (sum=0.0,j=1;j<=n;j++) sum += qt[j][i]*t[j];
            w[i]=fvec[i]-fvcold[i]-sum;
            if (fabs(w[i]) >= EPS*(fabs(fvec[i])+fabs(fvcold[i]))) skip=0;
            不要更新w中无噪的分量
            else w[i]=0.0;
        }
        if (!skip) {
            for (i=1;i<=n;i++) {                       t = Q^T · w.
```

```
                for (sum=0.0,j=1;j<=n;j++) sum += qt[i][j]*w[j];
                t[i]=sum;
            }
            for (den=0.0,i=1;i<=n;i++) den += SQR(s[i]);
            for (i=1;i<=n;i++) s[i] /= den;        在 s 中存贮 s/(s·s)
            qrupdt(r,qt,n,t,s);                      更新 R 和 Q^T
            for (i=1;i<=n;i++) {
                if (r[i][i] == 0.0) nrerror("r singular in broydn");
                d[i]=r[i][i];                        R 中对角元存入 d 中
            }
        }
    }
    for (i=1;i<=n;i++) {                  对于线性搜索计算 ∇f ≈ (Q·R)^T
        for (sum=0.0,j=1;j<=n;j++) sum += qt[i][j]*fvec[j];
        g[i]=sum;
    }
    for (i=n;i>=1;i--) {
        for (sum=0.0,j=1;j<=i;j++) sum += r[j][i]*g[j];
        g[i]=sum;
    }
    for (i=1;i<=n;i++) {                  存贮 x 和 F
        xold[i]=x[i];
        fvcold[i]=fvec[i];
    }
    fold=f;                               存贮 f
    for (i=1;i<=n;i++) {                  线性方程的右端是 -Q^T·F
        for (sum=0.0,j=1;j<=n;j++) sum += qt[i][j]*fvec[j];
        p[i] = -sum;
    }
    rsolv(r,n,d,p);                           存贮线性方程
    lnsrch(n,xold,fold,g,p,x,&f,stpmax,check,fmin);
    lnsrch 返回新的 x 和 f。当调用 fmin时，也在新的x处计算 fvec
    test=0.0;                                 测试函数值的收敛性
    for (i=1;i<=n;i++)
        if (fabs(fvec[i]) > test) test=fabs(fvec[i]);
    if (test < TOLF) {
        *check=0;
        FREERETURN
    }
    if (*check) {                             若线性搜索失败为真，以便寻找新的x
        if (restrt) FREERETURN                失败，已经试着重新初始化雅可比行列式
        else {
            test=0.0;                         检查 f 的梯度为零，即伪收敛
            den=FMAX(f,0.5*n);
            for (i=1;i<=n;i++) {
                temp=fabs(g[i])*FMAX(fabs(x[i]),1.0)/den;
                if (temp > test) test=temp;
            }
            if (test < TOLMIN) FREERETURN
            else restrt=1;                    试着重新初始化雅可比行列式
        }
    } else {                                  成功的步骤，对下一步将采用 Broyden 更新
        restrt=0;
        test=0.0;                             为 δx 收敛而测试
        for (i=1;i<=n;i++) {
            temp=(fabs(x[i]-xold[i]))/FMAX(fabs(x[i]),1.0);
            if (temp > test) test=temp;
        }
        if (test < TOLX) FREERETURN
    }
}
nrerror("MAXITS exceeded in broydn");
FREERETURN
}
```

### 9.7.3 更先进的实施

假如 J(在牛顿法中)或 B(在布罗依登法中)成为奇异的或接近奇异的,则至今所叙述的方法都可能失败,以使 $\delta x$ 不能被确定。倘若幸运的话,这种情况在实际中常常不会发生。解决处理这些问题的方法已有进展,所处理的问题包含若检测出奇异性或接近奇异性,则调节 J 的制约数及扰动 J。最容易实施的是,在牛顿法中用 *QR* 分解替代 *LU* 分解(参阅[1])。我们个人的经验是,当某一种方法,它能够解决 J 确实为奇异的问题以及标准牛顿法失效的问题;但是这方法应用于其他 *LU* 分解法成功的问题中,却可能稳健性减少。显然,这里重要的是要包含舍入误差,下溢等等问题。

我们的求极小化和寻找零值的全局策略都是基于线性搜索。其他一些全局算法,如**钩形步长**(*hook step*)法和**狗腿曲折形步长**(*dogleg step*)法,都是基于**模型可信赖域方法**,它和非线性最小二乘的勒温伯格——马阔特算法有关(第15.5节)。尽管这些方法比线性搜索法要复杂得多,但是这些方法以稳健性而盛名,甚至当起始时远离所期望的零值或最小值,它仍是稳健的[1]。

参考文献和进一步读物:

Dennis, J. E., and Schnabel, R. B. 1983, *Numerical Methods for Uncostrained Optimization and Nonlinear Equations*(Englewood Cliffs, NJ: Prentice-Hall). [1]

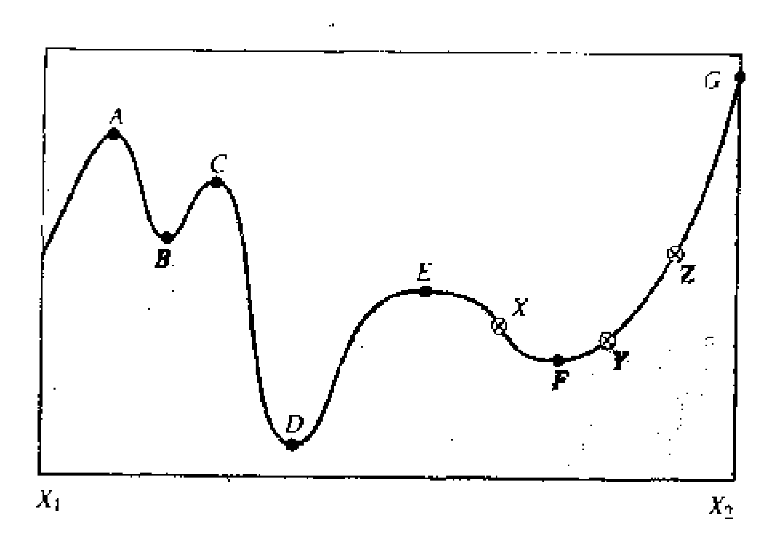
Broyden, C. G. 1965, Mathematics of Computation, vol. 19. pp. 577~593. [2]

# 第十章　函数的极值

## 10.0　引言

这一章我们所要研究的问题，简言之，即：给定一个函数 $f$，它依赖于一个或多个独立变量，问这些变量取何值时，$f$ 达到某一极大值或极小值。而在这个极大值点或极小值点处，函数 $f$ 的取值又是什么。求极大值问题与求极小值问题之间可以说是息息相关的，因为 $f$ 的极大值恰好与 $-f$ 的极小值相对应。对计算方法的要求，求极值与其它问题一样，都希望速度快、计算量小且占用内存少。一般说来，计算量的大小主要取决于函数值的计算量（根据算法的不同，可能还包括 $f$ 对其所有变元之偏导的数值计算量）。因此，我们设计算法的思想也很简单，即尽可能地使计算 $f$ 值的次数越少越好。

函数的极值（极大值或极小值）可以是整体的（即函数真正的最大值或最小值），也可能是局部的（在有限邻域内的最大值或最小值，不包括这个邻域的边界）（见图10.0.1）。一般说来，求整体极值是一个很难解决的问题，目前广泛采用的是两种标准的启发式方法：一是，从独立自变量的各种不同初值（象第7.7节中那样拟随机选取）开始，求出所有局部极值，然后从中选出最大的或最小的（如果这些局部极值不是完全一样的话）；二是，对局部极值以一有限步长的扰动，然后分析由此计算出来的结果，是稍有改进呢，还是"一直"保持不变。值得一提的是，近年来有一类称为"模拟退火"的方法，在各种全局极小化问题上取得了一些重要的和成功的进展（详见第10.9节）。




点 $A$、$C$、$E$ 为局部（非整体）极大值，点 $B$ 和 $F$ 为局部（不是整体的）极小值。整体极大值点为 $G$ 点，它位于区间的边界上，因此函数在该点的导数不必为零。整体极小值位于 $D$ 点。在点 $E$ 处导数值高于一阶的都为零，这种情况对某些算法来说，可能会带来处理上的不便。我们称点 $X,Y,Z$"划界"了极小值点 $F$，这是因为 $Y$ 点的函数值比 $X$ 点和 $Z$ 点的函数值都小。

图10.0.1　在某一区间内的函数极值

本章的题目也可以取作**最优化**，它是这一广泛的数值研究领域中所通常采用的名词。最优化方法之所以重要，是因为它所涉及的问题形形色色、种类繁多，而这一重要性在很大程度上也依赖于特殊问题的特殊要求。比如说，经济学家和工程师们着重关心的是**约束最优化问题**，这类问题对独立变量的允许值预先有一定的限制。例如，美国的小麦产量必须是一个非负的数等等。约束最优化问题中，研究得颇为透彻的一个领域称为**线性规划**，即目标函数及约束条件正好是独立变量的线性组合。本章的第10.8节将对线性规划中的所谓"单纯形法"进行讨论，但这一节的内容与本章其他各节有些不太连贯。第10.9节的内容也属于我们本章所讨论的主题之外的内容，其原因在于：所谓的"退火法"目前还很新，我们尚不能确定它适合于问题求解过程中的哪些步骤。但是，我们知道退火法确实解决了一些以前被认为在实际当中不能解决的问题，而且它直接提出了在大量局部极值中，求整体极值这类问题，其中这些局部极值的数目常常是不可预期的。

本章的其他各节，精选了几种无约束极小化问题的最优算法(为明确起见，在后面的讨论中，我们认为最优化问题就是求极小值问题)。这些章节是连贯的，后面的内容依赖于前面的讨论。如果读者只想找一种"完善"的算法来解决某个特殊问题，那么你可能会感到书中所讲述的内容比你想要了解的东西多。遗憾的是，目前还没有一种完善的最优化算法。因此我们建议读者多试验几种算法，多作些比较。对于最初算法的选择可以从如下几方面来考虑：

· 目前有两种方法可供选择，一种只需计算目标函数的值，另一种还需计算函数的导数值。在多维情况下，函数的导数即梯度，它是一个向量。使用导数的算法在某些功能上优于只使用函数值的算法，但功能上的优点通常不足以补偿所增加的导数的计算量。我们可以很容易地构造出这两种方法的例子。但是，如果函数的导数可以计算出来，则不妨试着将它们充分利用。

· 对于不需计算导数的一维极小值问题(即求一个自变量的函数的极小值)，可以先用第10.1节介绍的方法确定极小值点所在的区间，然后再用第10.2节的布伦特(Brent)方法求解。如果目标函数二阶(或更低阶)导数不连续，则布伦特方法的抛物线内插不适合使用，这时可使用**黄金分割搜索方法**的最简单的形式(详见第10.1节)。

· 对于需计算导数的一维极值问题，第10.3节提供了布伦特方法的一种变化形式，这种方法只是有限地利用了一阶导数的信息。一般情况下，我们都回避用导数信息去构造高阶的内插多项式，因为凭我们的经验，这种作法在收敛性方面的改进(即收敛到一个光滑的解析的极小值附近)，并不能弥补多项式在迭代初期作出错误内插的趋向，特别对于那些具有极明显"指数"特征的函数。

现在我们再来看看多维情况，讨论一下使用和不使用一阶导数的算法需注意的问题。

· 我们同样必须在两类方法之间做选择，一种是存储数量级为 $N^2$ 的算法，另一种是数量级仅为 N 的算法(这里 $N$ 为维数)。虽然对于 $N$ 的适中值以及合理的内存大小，这一约束并不算苛刻，但偶尔也有一些问题对存储量的要求很严格。

· 在第10.4节中，我们将给出**下降的单纯形法**，这种不太受人重视的算法是由内尔德(Nelder)和米德(Mead)提出的。(注意，这里所用的"单纯形"一词不能与线性规划中的单纯

形方法相混淆。)它对目标函数几乎不需作任何特殊的假设,是一种直接的下降算法。其下降速度虽然很慢,但在某些情况下,也算是一种功能很强的算法。特别值得一提的是,这种算法的源程序相当简明,而且完全不需其他任何调用。你也许会觉得不可思议。这个求解一般 $N$ 维极小化问题的程序还不是100行语句!如果在某个整体项目中,求极小值问题仅仅是其中的一个枝节部分,那么这种方法将是最实用的,其存储数量级为 $N^2$,并且不需计算导数

· 第10.5节将介绍以**鲍威尔(Powell)算法**为原型的**方向集法**,在函数的导数不易计算时,可选择这类方法,即使在导数可以计算时,读者也不妨一试。方向集法虽然不需要计算导数,但却要用到一维极小化子算法,如布伦算法等(见上文)。这种方法的存储数量级也达到 $N^2$。

在求多维极小化问题的算法中,需要一阶导数参加运算的方法主要有两类,它们都要利用一维的子算法,而子算法本身用不用导数信息都无妨,只要认为合适即可(依赖于函数及其梯度向量的相对计算量)。需要说明的是,对于各种应用问题来说,并不是某一类算法占主导地位,两者均可供选择:

· 第一类通称为**共轭梯度算法**,典型的有**弗莱彻-里弗斯(Fletcher-Reeves)算法**,以及与之密切相关且比它有所改进的**波拉克-里拜尔(Polak-Ribiere)算法**。共轭梯度法的存储量只是 $N$ 的几倍数量级,但需要计算导数及一维子算法。详细的讨论及程序实施请见第10.6节。

· 第二类算法统称为**拟牛顿(Newtow)法**或**变度量法**,典型的有戴维登-弗莱彻-鲍威尔(Davider-Fletcher-Powell)算法(简写为 DFP 算法,有时也称 Flecher-Powell 算法、布罗依顿-弗莱彻-戈尔法布-香农算法(BFGS)等)。这些算法存储量都达到 $N^2$,需要导数信息及一维子算法,详见第10.7节。

至此,我们就可以进入本章正题。试探一下求极小值问题的难易程度如何。

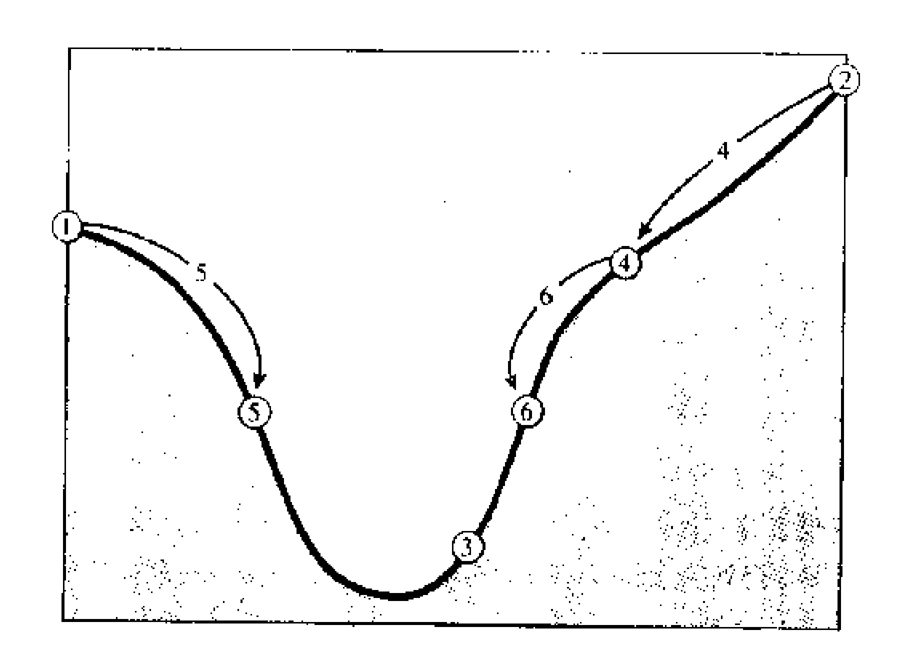
## 10.1　一维黄金分割搜索

让我们回忆一下,第9.2节中的二分法在一维情况下是如何求解方程根:首先,假设根落在某一区间$(a,b)$内,然后,计算函数在区间中间某点 $x$ 的值,得到一个较小的划界的区间$(a,x)$和$(x,b)$。其长度显然小于原区间$(a,b)$。二分法求根即对上述过程进行迭代,直到区间的长度足够小为止。显然将 $x$ 选为$(a,b)$区间的中点是最恰当的,这样即使在函数"不合作"的情况下(即下一解区间总是被迫选为较长的区间段),也能使区间长度的减小速度达到最快。

二分法的这些思想完全可以平移到求极小值问题上,但有一点不太好捉摸的是:如何对极小值点划界呢?如果函数在 $a,b$ 两点(满足 $a<b$)的值具有相反的符号,则该函数必有一个根落在点对 $a$ 与 $b$ 之间。与之对照,如果当有三点组 $a,b,c$ 满足 $a<b<c$ 或 $c<b<a$,且 $b$ 点函数值比 $a$ 点和 $c$ 点的值都小,则称为有一极小值被划界。对上面这种情况,函数(如果是非奇异的)极小值显然落在区间$(a,c)$内。

与二分法类似,我们需要在 $a$ 和 $b$ 之间或 $b$ 和 $c$ 之间选择一个新的点 $x$。假定在 $b$ 和 $c$ 之间选定了某点 $x$,并计算出 $f(x)$的值。如果 $f(b)>f(x)$,则取新的三点组为$(a,b,x)$;相

反地，如果 $f(b)<f(x)$，则新的三点组将取为 $(b,x,c)$。不论哪一种情况，三点组的中间点的纵坐标总是当前得到的函数值中最小的，见图10.1.1。继续这一过程，直至三点组中的两外点距离小到可容许的程度为止。




包含极小点的划界区间开始时取为点1,3,2。计算函数在点4的值，由点4代替点2；然后由点5代替点1；接着又由点6代替点4。每步迭代的规则是保持三点组的内点函数值始终比两外点的函数值小。经过图中所示的几步搜索之后，极小值点被划界在点3,6,5之间。

图10.1.1　求极小值的逐步划界过程

但"可容许小"到底有多小呢？若函数在 $b$ 点取极小值，读者可能会简单地认为，极小值点的划界区间范围最终可以小到 $(1-\epsilon)b<b<(1+\epsilon)b$，这里 $\epsilon$ 为所使用的计算机的浮点精度，如 $3\times10^{-8}$（对浮点数）或 $10^{-15}$（对双精度数）。事实上不是这样的！一般地，在 $b$ 点附近 $f(x)$ 的形式可由泰勒(Taylor)定理给出：

$$f(x)\approx f(b)+\frac{1}{2}f''(b)(x-b)^2 \tag{10.1.1}$$

当

$$|x-b|<\sqrt{\epsilon}\,|b|\sqrt{\frac{2|f(b)|}{b^2f''(b)}} \tag{10.1.2}$$

时式(10.1.1)中第二项较第一项小 $\epsilon$ 倍，若将它加到和式中无异相当于添零，因此可忽略。我们将式(10.1.2)中的右端项写成所示形式是因为，对绝大多数函数来说，最后的平方根项是一个单位阶的数。因此，根据粗略估计，要想划界一个解区间，使其长度小于其中心值的 $\sqrt{\epsilon}$ 倍，即只有大约 $10^{-4}$（单精度）或 $3\times10^{-8}$（双精度）的宽度，这几乎是不可能的。只要我们清楚这一事实，就可以避免做许多无用功。

本章给出的求极小值的程序通常需要——由用户提供一个参数 tol，并要求程序返回横坐标的相对精度达到 ±tol（划界区间的长度为 2×tol）。除非能对式(10.1.2)的右端项给

出更好的估计值，否则必须置 tol 等于机器浮点精度的平方根(不能小得太多)，因为比之更小的值将得不到任何结果。

剩下的问题就是要在给定 $a,b,c$ 的条件下，如何确定新的点 $x$。设 $b$ 分 $a,c$ 间的距离为比例 $w$，即：

$$\frac{b-a}{c-a}=w \qquad \frac{c-b}{c-a}=1-w \tag{10.1.3}$$

又设下一试验点 $x$ 与 $b,c,a$ 的关系如下式所示：

$$\frac{x-b}{c-a}=z \tag{10.1.4}$$

则下一个划界段的长度将是当前区间长度的 $w+z$ 倍或 $1-w$ 倍。为了使最坏情况出现的可能性达到最小，我们选择一个 $z$ 值，使得 $w+z=1-w$，即

$$z=1-2w \tag{10.1.5}$$

立即可以看出，新点 $x$ 是在原区间内 $b$ 点的对称点，也就是说 $|b-a|$ 等于 $|x-c|$。这说明点 $x$ 位于两小段区间中较长的那一段中(仅当 $w<\frac{1}{2}$ 时，$z$ 为正)。

但是 $x$ 究竟位于较长区间段的什么具体位置上？$w$ 值本身又从哪里得到呢？我们假设 $w$ 是从前面的步骤中得到的。因此，如果 $z$ 的选择是最佳的，那么在此以前选择的 $w$ 也是最佳的。这一比例相似性说明 $x$ 分 $b,c$(假设从 $b$ 到 $c$ 为较长区间段)的比例与 $b$ 分 $a,c$ 的比例应相同，也就是说：

$$\frac{z}{1-w}=w \tag{10.1.6}$$

由(10.1.5)式及(10.1.6)式可以得到二次方程

$$w^2-3w+1=0 \quad 解得 \quad w=\frac{3-\sqrt{5}}{2}\approx 0.38197 \tag{10.1.7}$$

从这一结论分析，最佳划界区间 $a<b<c$ 应满足内点 $b$ 与其中一个外点(如 $a$)的距离比例为 0.38197，而与另一外点(如 $c$ 的距离比例为 0.61803 。这些比例即所谓的**黄金中项**或**黄金分割数**，它的美学性质可追溯到遥远古代的毕达哥拉斯年代(Pythagoreans)，这种与二分法求根类似的函数极小化最优方法也由此得名为**黄金分割搜索**。我们将其归纳如下：

在每一步划界中给定一个三点组，下一试探点取在较长区间段的 0.38197 分割点(从三点组的中间点量起)。如果开始时划界的三点组的各段没有按照黄金曲率分割，那么逐次选择黄金分割点的程序将会很快地收敛到合适的，自行重复的比率。

黄金分割搜索可以保证每计算一次函数值(获得自行重复的比率之后)，划界区间长度变为原来 0.61803 倍，但这与二分法求根中的 0.50000 倍相比，显然要差一些。黄金分割法的收敛是线性的(按第九章的语言来说)，即随着附加函数值的计算，有效数字逐步线性增长。下一节中我们还将介绍一种超线性方法，这种方法每作一次函数计算，有效数字的增长速度都有所提高

### 10.1.1 确定初始划界为极小的程序

我们在前面的讨论中，已假定初始划界区间是能够确定的，并认为这是所有一维极小值问题的一个基本步骤。虽然有某些一维算法不需要确定严格的初始划界的区间，但是即使这

些非划界算法程序在运行过程中能保证函数值不断减小，我们也始终不能确信极小值的存在。因此在求极小值(或零解)之前，一定要确定包含它们的区间！

确定初始划界的步骤和方法没有太多的理论依据。虽然向下搜索这一方向是很明确的，但究竟搜索多远呢？从初始预测值(胡猜的？)开始，我们希望采取越来越大的步长，加大步长的方法既可以通过一常数因子，也可以利用前面的点作抛物线外推的结果，这种外推的设计就是为了使我们达到外推拐点。即使步长增加到很大也是没有多大关系的，因为我们毕竟在作向下搜索，所以已经具备了三点划界点的左边点和中间点，再用足够大的一步以停止下降的趋势，就可以得到函数值略高的第三个点。

确定初始搜索区间的标准程序如下所示：

```
#include <math.h>
#include "nrutil.h"
#define GOLD 1.618034
#define GLIMIT 100.0
#define TINY 1.0e-20
#define SHFT(a,b,c,d) (a)=(b);(b)=(c);(c)=(d);          其中GOLD是连续区域放大的缺省倍率;GLIMIT
                                                         是抛物线拟合过程中允许的最大放大倍数。

void mnbrak (float *ax, float *bx, float *cx, float *fa, float *fb, float *fc, float (*func)(float))
    给定一个函数func及两个相异初始点ax和bx,本程序沿向下方向进行搜索(方向是由函数在初始点处的值确定),
    并返回三个新的点ax,bx,cx,由它们确定的区间能划界函数的一个极小值;本程序同时返回的还有函数在这三个
    点的值fa,fb及fc。
{
    float ulim,u,r,q,fu,dum;

    *fa=(*func)(*ax);
    *fb=(*func)(*bx);
    if (*fb > *fa) {                          交换a、b的位置，使能够沿着由a到b方向向下
        SHFT(dum,*ax,*bx,dum)                    搜索
        SHFT(dum,*fb,*fa,dum)
    }
    *cx=(*bx)+GOLD*(*bx-*ax);                 c的第一个估值
    *fc=(*func)(*cx);
    while (*fb > *fc) {                       在界定搜索区间之前，使程序在此继续返回通
        r=(*bx-*ax)*(*fb-*fc);                过a、b、c的抛物外推计算u
        q=(*bx-*cx)*(*fb-*fa);                       TINY在此处的作用是防止被零除
        u=(*bx)-((*bx-*cx)*q-(*bx-*ax)*r)/        的情况发生
            (2.0*SIGN(FMAX(fabs(q-r),TINY),q-r));
        ulim=(*bx)+GLIMIT*(*cx-*bx);
        至此程序不能再往下进行了。现在来考虑一下各种可能的情况
        if ((*bx-u)*(u-*cx) > 0.0) {           u位于b和a之间
            fu=(*func)(u);
            if (fu < *fc) {                    在b、c之间找到了一个极小点
                *ax=(*bx);
                *bx=u;
                *fa=(*fb);
                *fb=fu;
                return;
            } else if (fu > *fb) {             在a和u之间找到了一个极小点
                *cx=u;
                *fc=fu;
                return;
            }
            u=(*cx)+GOLD*(*cx-*bx);            抛物线拟合不发挥作用利用缺省的放大倍数
            fu=(*func)(u);
        } else if ((*cx-u)*(u-ulim) > 0.0) {       抛物线拟合在c及其允许限之间
            fu=(*func)(u);                              进行
```

```
            if (fu < *fc) {
                SHFT(*bx,*cx,u,*cx+GOLD*(*cx-*bx))
                SHFT(*fb,*fc,fu,(*func)(u))
            }
        } else if ((u-ulim)*(ulim-*cx) >= 0.0) {      将抛物线 u 界定在最大允许值内
            u=ulim;
            fu=(*func)(u);
        } else {                                      放弃 u，利用缺省的放大倍数
            u=(*cx)+GOLD*(*cx-*bx);
            fu=(*func)(u);
        }
        SHFT(*ax,*bx,*cx,u)                           去掉最开始的点，继续执行搜索
        SHFT(*fa,*fb,*fc,fu)
    }
}
```

（因为上述程序要对三、四个点及其函数值进行分析和讨论，所以其结尾看起来似乎很麻烦；本章中还有几个程序也是如此，但它们中间包含的思想却相当简单。

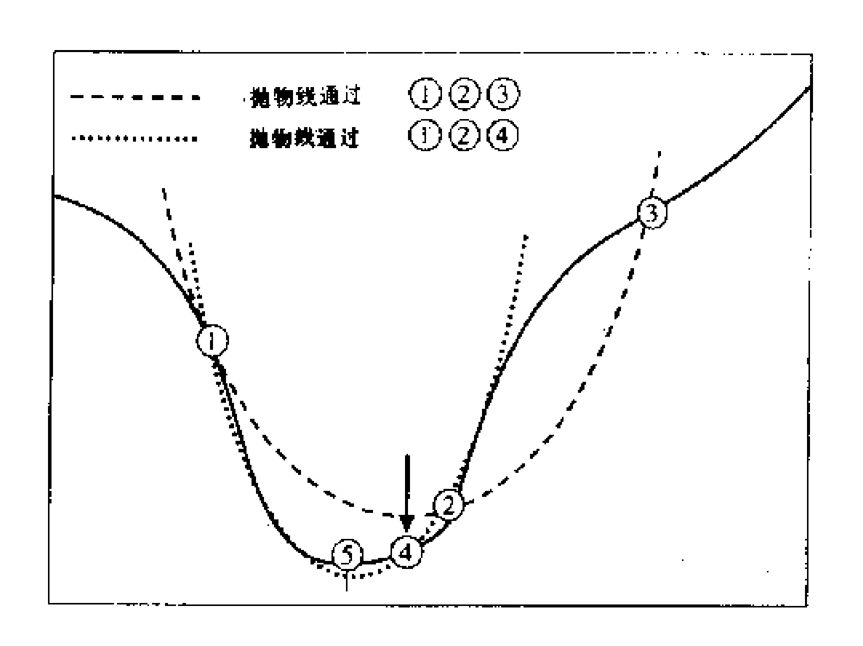
## 10.1.2 黄金分割搜索方法的应用程序

```
#include <math.h>
#define R 0.61803399                     黄金分割率
#define C (1.0-R)
#define SHFT2(a,b,c) (a)=(b);(b)=(c);
#define SHFT3(a,b,c,d) (a)=(b);(b)=(c);(c)=(d);

float golden(float ax, float bx, float cx, float (*f)(float), float tol, float *xmin)
    给定一个函数 f 及一个包含极小值的三点组 ax,bx,cx(其中 bx 位于 ax 与 cx 之间，其函数值 f(bx)比 f(ax)和 f(cx)
    都小)，本程序将利用黄金分割搜索方法求 f 的极小值，结果的相对精度可达到 tol，其中极小值点的横坐标将以
    xmin 返回，而极小点的函数值则以 golden 返回(golden 也是本程序的返回值)。
{
    float f1,f2,x0,x1,x2,x3;
    x0=ax;                                  每个时刻都要跟踪四个点
    x3=cx;                                      x0,x1,x2,x3.
    if (fabs(cx-bx) > fabs(bx-ax)) {        以x0，x1 作为较短区间段的端点
        x1=bx;
        x2=bx+C*(cx-bx);                    并填入新的测探点
    } else {
        x2=bx;
        x1=bx-C*(bx-ax);
    }
    f1=(*f)(x1);                            函数值的计算。注意函数在原始端点的值总是
    f2=(*f)(x2);                                不需计算
    while (fabs(x3-x0) > tol*(fabs(x1)+fabs(x2))) {
        if (f2 < f1) {                      一个可能的结果
            SHFT3(x0,x1,x2,R*x1+C*x3)       作记录
            SHFT2(f1,f2,(*f)(x2))           计算新的函数值
        } else {                            另一种结果
            SHFT3(x3,x2,x1,R*x2+C*x0)
            SHFT2(f2,f1,(*f)(x1))           计算新的函数值
        }
    }                                       若进行完毕则回头看
    if (f1 < f2) {                          进行完毕。输出两个当前值中最佳的那一个
        *xmin=x1;
        return f1;
    } else {
        *xmin=x2;
        return f2;
    }
}
```

## 10.2 抛物线内插和一维布伦特方法

在上一节给出的程序 **mnbrak** 中，曾经提到过抛物线内插用途，现在我们来作进一步的详细探讨。假设我们费了半天劲才发现某个极小点，而一直未计算出来，这不能不令人恼火；实际上黄金分割搜索方法的设计正是为了有效地处理这种最坏的可能。但我们为什么要假设这种最坏情况呢？如果函数在极小值附近具有很好的抛物线性质（足够光滑的函数一般都具备这一性质），那么利用过任意三点拟合出来的抛物线，应能够很容易地求出极小值，或者至少能够非常接近极小值（见图10.2.1）。由于我们的目标是求极小值点的横坐标，而不是其纵坐标，因此上述方法在技术上被称为**逆抛物线内插**。




过给定函数（图中以实线表示）上的三点1，2，3作一条抛物线（图中虚线所示），计算在抛物线的极小值点处的函数值4，并以此点代替点3。再过点1，4，2作一条新的抛物线（图中点线所示），这条抛物线的极小值点5已经很接近目标函数的极小值点。

图10.2.1　用逆抛物线内插法收敛到极小点的过程图

对于过 $f(a)$，$f(b)$，$f(c)$ 三点的抛物线，其极小值点的横坐标公式为：

$$x = b - \frac{1}{2}\frac{(b-a)^2[f(b)-f(c)]-(b-c)^2[f(b)-f(a)]}{(b-a)[f(b)-f(c)]-(b-c)[f(b)-f(a)]} \tag{10.2.1}$$

显然 $x$ 能很容易得到。只有当三点共线，即分母为零时（此时抛物线的极小值在无穷远处），这个公式才是无意义的。但是，还应注意到由式（10.2.1）求出的点既可以是抛物线的极小值，也可能是极大值点。在实际当中，没有一种求极小值的方案仅仅依赖于式（10.2.1）就能得到成功的结果。

我们的任务就在于设计一种方案，当函数“不配合”时能依赖于一种收敛速度慢但很可靠技巧（比如黄金分割搜索方法）来求解；而在目标函数允许的情况下，又能转而利用式（10.

2.1)来求解。看来这个任务并不轻松，原因是i)在两种方法间转换时，为避免不必要的函数计算而所需的记录工作可能会很复杂；ii)在计算结果的末尾小数时必须谨慎小心，因为此时函数值的计算已经非常接近方程(10.2.1)的舍入极限。iii)检验目标函数配合与否的方案设计必须要考虑周全。

**布伦特方法**(Brent[1])能在各种特殊情况下完成上述任务，它在每一特定阶段跟踪记录总共六个函数点(不必全不相同) $a,b,u,v,w$ 及 $x$，这些点的定义如下：$a,b$ 之间包含极小值点；$x$ 是当前所能找到的最小函数值对应的点(或在函数值相等的情况下，$x$ 是最新求出的点)；$w$ 点的函数值为第二小的函数值；$v$ 是 $w$ 的上一次迭代值；而最近一次计算函数值是在 $u$ 点。布伦特法中，还将出现点 $x_m$，它是$(a,b)$的中间点，但不在该点计算函数值。

阅读下文的程序可以了解算法的逻辑结构，但这里还要提几条有益的一般性原则。假设用抛物线内插拟合点 $x$、$v$ 及 $w$。为了达到求解的目的，抛物线必须满足(i)在限定区间$(a,b)$内下降，(ii)从当前最佳值 $x$ 移动的步长能够小于前面第二步移动步长的一半。后一条准则保证了内插步骤可以有效地收敛到某点，而不会在某一非收敛的有限范围内跳跃。在最坏可能的情况，抛物线内插可以进行但不起作用，这时布伦特方法将交替使用内插和黄金分割两种方法，并由后者保证其收敛性。在算法中要与前面第二步的结果进行比较的理由是：经验表明，如果某一步迭代结果不佳，但却能在下一步中得到补偿，这时最好不要对其进行"惩罚"这一点从本质上来看是很有启发性的。

程序中还有一条限制是，函数在与已计算过的点(或已知的端点)相距不足 tol 的点处不需计算其值，其中的原因正如我们在式(10.2.3)中所看到的，这样做没有任何信息包含在里面：这些点的函数值之间的差距仅仅在舍入误差的数量级上，因此在下面的程序中读者可以看到，有几处对新的可能试验点的检验和调整反映了这条限制。此外，这条限制随着"已做点"的测试会有细微的相互影响。因此这点也同样应考虑在列。

布伦特方法的运行结果，通常是 $a$ 与 $b$ 之间相距$2*x*$tol，其中 $x$(极小值点的横坐标)为 $a$ 与 $b$ 的中点，因此它的结果精度将达到±tol。

最后应提醒大家注意的是 tol 一般不能小于机器浮点精度的平方根。

```
#include <math.h>
#include "nrutil.h"
#define ITMAX 100                    允许迭代的最大次数
#define CGOLD 0.3819660              黄金分割率
#define ZEPS 1.0e-10                 对恰巧为零的极小值情况而设置的一个相对精确度的小数
#define SHFT(a,b,c,d) (a)=(b);(b)=(c);(c)=(d);

float brent(float ax, float bx, float cx, float (*f)(float), float tol, float *xmin)
    给定一个函数f并给定划界三点的横坐标ax,bx,cx(满足bx位于ax与cx之间,其函数值f(bx)比f(ax)和f(cx)都
    小)。本程序将用布伦特方法搜索f的极小值点,结果的相对精度可达tol,其中横坐标以xmin返回,函数值以brent
    返回(这也是本程序的返回值。)
{
    int iter;
    float a,b,d,etemp,fu,fv,fw,fx,p,q,r,tol1,tol2,u,v,w,x,xm;
    float e=0.0;                                    前第二步移动的距离

    a=(ax < cx ? ax : cx);                          不管a、b的输入情况如何，其横坐标须
    b=(ax > cx ? ax : cx);                              按上升顺序
    x=w=v=bx;                                       初始化
    fw=fv=fx=(*f)(x);
```

```
    for (iter=1;iter<=ITMAX;iter++) {              主循环
        xm=0.5*(a+b);
        tol2=2.0*(tol1=tol*fabs(x)+ZEPS);
        if (fabs(x-xm) <= (tol2-0.5*(b-a))) {      这里做测试
            *xmin=x;
            return fx;
        }
        if (fabs(e) > tol1) {                      构造一个试探用的抛物线拟合
            r=(x-w)*(fx-fv);
            q=(x-v)*(fx-fw);
            p=(x-v)*q-(x-w)*r;
            q=2.0*(q-r);
            if (q > 0.0) p = -p;
            q=fabs(q);
           etemp=e;
           e=d;
           if (fabs(p) >= fabs(0.5*q*etemp) || p <= q*(a-x) || p >= q*(b-x))
               d=CGOLD*(e=(x >= xm ? a-x : b-x));
           上述条件决定抛物线内插是否合适。在较长区间段采用黄金分割法

           else {
               d=p/q;                              采用抛物线内插
               u=x+d;
               if (u-a < tol2 || b-u < tol2)
                   d=SIGN(tol1,xm-x);
           }
       } else {
           d=CGOLD*(e=(x >= xm ? a-x : b-x));
       }
       u=(fabs(d) >= tol1 ? x+d : x+SIGN(tol1,d));
       fu=(*f)(u);
       每次迭代在此处计算一次函数值
       if (fu <= fx) {                             并判断下一步该做什么，接下来内务
           if (u >= x) a=x; else b=x;                操作
           SHFT(v,w,x,u)
           SHFT(fv,fw,fx,fu)
       } else {
           if (u < x) a=u; else b=u;
           if (fu <= fw || w == x) {
               v=w;
               w=u;
               fv=fw;
               fw=fu;
           } else if (fu <= fv || v == x || v == w) {
               v=u;
               fv=fu;
           }
       }                                           整理完毕。返回进入下轮迭代
    }
    nrerror("Too many iterations in brent");
    *xmin=x;
    return fx;
}
```

参考文献和进一步读物：


Brent, R. P. 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 5. [1]


## 10.3 使用一阶导数的一维搜索方法

在这一节中我们将详细讨论与上节同样的问题，即在具有横坐标 $a<b<c$ 划界的三点

中搜索函数的极小值。所不同的是，除利用函数本身的值之外，本节所介绍的方法还要利用函数的一阶导数信息。

一般情况下，我们可能只是简单地想到利用象程序（如第9.2节的 **rtflsp** 或 **zbrent**）中的求根的方法，来搜索导数的零点，而将函数值的信息撇开。但很快我们就会放弃这一想法：因为这样做根本无法区分是极大值，还是极小值；而且在初始条件下，如果界点的导数信息说明"下降"应沿划界区间之外的方向，那么从这样的初始条件来分析我们根本无从下手。

但我们在任何时候都不想放弃，将极小值严格划界在某一区间内这一策略。保持这个区间的唯一途径就是用函数（而不是导数）信息不断更新它，使其中心点的函数值保持最低。因此，导数起的角色仅仅是帮助我们在区间内选择新的试验点而已。

有一种观点认为"应利用已经得到的一切信息"，即求一个相对高阶的多项式（三次或更高次），使其满足先前得到的一些函数值和导数值。例如，满足两点的函数值和导数值的三次多项式是唯一的，因此我们可以转而求这个三次多项式的内插极小值（如果在划界的区间内存在极小值的话）。上述方法的计算公式已由戴维登（Davidon）等人导出，详见[1]。

与上述观点相比，本书中的方法略保守。一旦一种算法确定为超线性收敛，其收敛阶数低一些或高一些都是无关紧要了。在我们所遇到的一些实际问题当中，绝大多数函数计算都花在力图使算法足够接近极小值，以确保超线性收敛上。所以我们更关心的是，高阶多项式可能引起各种麻烦（参见图3.0.1b）以及它们对舍入误差的敏感程度。

因此，我们只能这样利用导数信息：根据在三个划界 $a<b<c$ 之中心点的导数符号唯一地决定下一试验点是取在 $(a,b)$ 内，还是在 $(b,c)$ 内。将这个导数值以及当前仅次于极小值点的导数值用弦截法（逆线性插值）外推到零（注意弦截法本身是超线性收敛的，其收敛阶为1.618，这又是一处黄金分割比率，详见[1]57页）。对新的试验点，我们也象在布伦特方法中那样加上同样的限制。如果该试验点必须被放弃，则对区间进行精细二分。

在一维极小化问题中利用导数信息时我们的确是很保守的。但是，由于某些函数计算出来"导数"并不能与函数值相统一，也不能精确地指向求极小值的路径，这种情况不能不令人恼火，它频繁出现的原因常常是因为舍入误差的影响，有时也是由于在计算导数的方法中出现截断误差。

我们将会看到，下面的程序与上节的程序 **brent** 在形式上极为相似。

```
#include <math.h>
#include "nrutil.h"
#define ITMAX 100
#define ZEPS 1.0e-10
#define MOV3(a,b,c,d,e,f)(a)=(d);(b)=(e);(c)=(f);

float dbrent(float ax, float bx, float cx, float (*f)(float), float (*df)(float),
    float tol,float *xmin)
```
给定一个函数 f 及其导函数 df，并给定一个三个划界点的横坐标 ax，bx，cx（满足 bx 位于 ax 与 cx 之间，且 f(bx) 小于 f(ax) 及 f(cx)）。本程序将用布伦特方法的修改形式，即利用导数的布伦特法搜索出 f 的极小值，结果的相对精度可达 tol。极小值横坐标以 xmin 返回，最小函数值以 dbrent 返回，(dbrent 也是本程序的返回值)。
```
{
    int iter,ok1,ok2;                        作为标志，用于记录提议的方法是否可
    float a,b,d,d1,d2,du,dv,dw,dx,e=0.0;     接受
    float fu,fv,fw,fx,olde,tol1,tol2,u,u1,u2,v,w,x,xm;
```

下面的注释将指出与程序brent的不同之处，读者应首先阅读该程序 routine first.

```
a=(ax < cx ? ax : cx);
b=(ax > cx ? ax : cx);
x=w=v=bx;
fw=fv=fx=(*f)(x);
dw=dv=dx=(*df)(x);                              整理的信息既包括数值也包括函数值
for (iter=1;iter<=ITMAX;iter++) {
    xm=0.5*(a+b);
    tol1=tol*fabs(x)+ZEPS;
    tol2=2.0*tol1;
    if (fabs(x-xm) <= (tol2-0.5*(b-a))) {
        *xmin=x;
        return fx;
    }
    if (fabs(e) > tol1) {
        d1=2.0*(b-a);                           将d设置为搜索区间之外的值
        d2=d1;
        if (dw != dx) d1=(w-x)*dx/(dx-dw);      弦截法分别作用于两个点
        if (dv != dx) d2=(v-x)*dx/(dx-dv);
        到底取哪个d估值？我们的原则是它们位于搜索区间内，且在x的导数所指的一侧

        u1=x+d1;
        u2=x+d2;
        ok1 = (a-u1)*(u1-b) > 0.0 && dx*d1 <= 0.0;
        ok2 = (a-u2)*(u2-b) > 0.0 && dx*d2 <= 0.0;
        olde=e;                                 前第二步的移动步长
        e=d;
        if (ok1 || ok2) {                       只取一个d值，若两个都可以，取最小
            if (ok1 && ok2)                         的那个
                d=(fabs(d1) < fabs(d2) ? d1 : d2);
            else if (ok1)
                d=d1;
            else
                d=d2;
            if (fabs(d) <= fabs(0.5*olde)) {
                u=x+d;
                if (u-a < tol2 || b-u < tol2)
                    d=SIGN(tol1,xm-x);
            } else {                            二分法而不是黄金分割法
                d=0.5*(e=(dx >= 0.0 ? a-x : b-x));
                由导数的符号决定取哪个区间段
            }
        } else {
            d=0.5*(e=(dx >= 0.0 ? a-x : b-x));
        }
    } else {
        d=0.5*(e=(dx >= 0.0 ? a-x : b-x));
    }
    if (fabs(d) >= tol1) {
        u=x+d;
        fu=(*f)(u);
    } else {
        u=x+SIGN(tol1,d);
        fu=(*f)(u);
        if (fu > fx) {                          若在下降过程中反而出现上升则已找
            *xmin=x;                                到极小值点
            return fx;
        }
    }
    du=(*df)(u);                                现在对全部信息进行整理
    if (fu <= fx) {
        if (u >= x) a=x; else b=x;
        MOV3(v,fv,dv, w,fw,dw)
        MOV3(w,fw,dw, x,fx,dx)
        MOV3(x,fx,dx, u,fu,du)
    } else {
```

```
            if (u < x) a=u; else b=u;
            if (fu <= fw || w == x) {
                MOV3(v,fv,dv, w,fw,dw)
                MOV3(w,fw,dw, u,fu,du)
            } else if (fu < fv || v == x || v == w) {
                MOV3(v,fv,dv, u,fu,du)
            }
        }
    }
    nrerror("Too many iterations in routine dbrent");
    return 0.0;                                    永远不到达这里
}
```

参考文献和进一步读物：


Acton, F. S. 1970, *Numerical Methods That Work*, 1990, corrected edition (Washington: Mathematical Association of America), pp. 55; 454~458. [1]


## 10.4 多维下降单纯形法

从这一节开始我们讨论多维极小化问题，即求有多个独立变元的函数之极小值。但本节的内容与后面各节的内容相差较远，本节之后介绍的所有算法都将利用第10.1、10.2、10.3节的一维极小化算法作为其求解策略的一部分，而本节的算法完全是封闭式的，不需插入使用一维算法。

**下降的单纯形法**由内尔德(Nelder)和米德(Mead)提出(见文献[1])，这种方法只需计算函数值，不需导数参加运算。就其所需的函数值计算的次数而言，该方法的效率不算很高。而第10.5节的鲍威尔(Powell)算法(第10.5节)对所有可能的应用问题，其求解速度几乎肯定快得多。但是，对于一个计算量小的问题，并且函数图形的趋势可使计算速度变快些，这时下降单纯法常常是最佳选择的方法。

下降单纯形法所具有的几何特性，能使这方法描述起来或读起来都易理解，我们将其介绍如下：

一个**单纯形**是一个几何形体，它在 $N$ 维情况下，是由 $N+1$个点(或顶点)、所有相互连接的线段以及多边形面等等组成的几何图形。二维情况单纯形即为三角形；在三维情况，则为一个四面体，但不一定必须是规则的四面体。(线性规划中的单纯形方法也利用单纯形的几何概念，但是它与本节讨论的算法没有任何的关联。)一般情况下，我们只研究非退化的单纯形，它包含一个有限的 $N$ 维内体积。若取其上任意一点作为原点，则剩下的 $N$ 个点定义了跨越 N 维向量空间的向量方向。

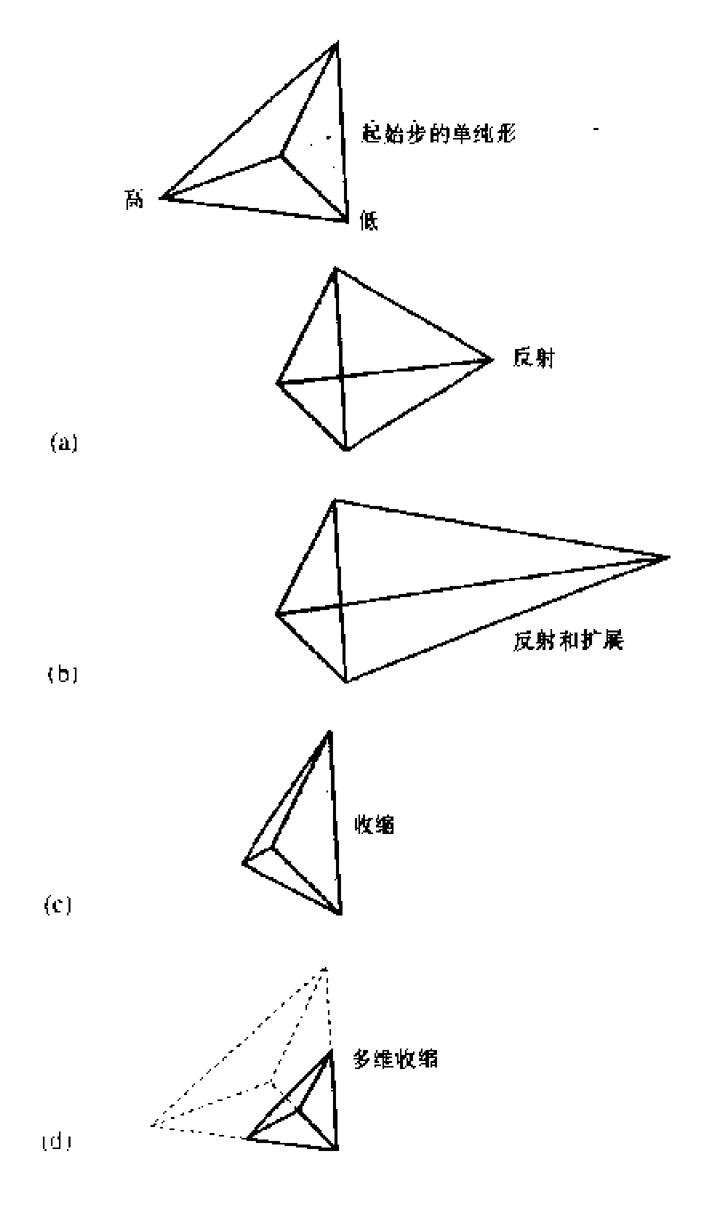
在一维情况可以划界极小值的区间，从而能保证迭代求解的成功。可惜在多维空间中没有类似的方法可寻。在多维情形，我们所能做到的最多就是给出一个初始预测值，即由独立变量组成的一个 $N$ 维向量作为迭代起点，然后设计一种算法不断地作下降搜索，直到遇到一个(至少是局部)极小值为止，而在搜索期间 $N$ 维形体操作的复杂程度将是难以想象的。

下降的单纯形法必须从 $N+1$个点而不仅仅是从单个点开始迭代，这 $N+1$个点定义了一个初始单纯形，如果把其中一个点(哪一个无关紧要)作为初始起点 $\mathbf{P}_0$，则其他 $N$ 个点可取为：

$$\mathbf{P}_i = \mathbf{P}_0 + \lambda\,\mathbf{e}_i \tag{10.4.1}$$

其中 $e_i$ 为 $N$ 个单位向量。$\lambda$ 为一常数，它是对问题的特征长度大小的估计值。(也可以对每一个向量方向取不同的 $\lambda_i$ 值。)

然后下降的单纯形法将采取一系列步骤。其中绝大多数步骤是，将函数值达到最大的单纯形点(即"最高点")通过单纯形的背向面移到一个较低点。这些步骤称为反射，构造它们的目的是为了保持单纯形的体积不变(从而保持其非退化性)。之后，下降的单纯形法将对单纯形在某个方向上进行扩展以加大步长。当到达"谷底"处时，单纯形将自行作横向收缩，并试图流入谷地。如遇到单纯形需"穿过针眼"的情况，它会从各个方向自行收缩，且自行拉向最低点(即最佳点)附近。程序 **amoeba** 的取名即描述这一过程；图10.4.1概括了几种基本的拉伸动作。




在这一步开始时，单纯形(这里是四面体)如顶图所示。在这一步结束之后，单纯形可变为一个：(a)从高点的背向反射，(b)从高点的背向反射并扩展，(c)从高点沿一维的收缩，(d)沿各个方向向低点的收缩，算法经过许多步这样的迭代之后，一般总能收敛到函数的极小值。

图10.4.1 下降单纯形法中某步的几种可能的结果

在多维极小程序中，终止准则的确定可能是很棘手的问题。由于没有划界区间，加上独立变量又不只一个，我们不可能象一维那样只要对一个独立变元给出某一容差值就可以了。多维情况下一般要在算法中确定一个“循环”或者一个“步骤”，这样在那一步中，当向量移动的距离大小上小于某一容差值 tol 时，算法才有可能终止。当然也可以将终止准则定义为函数值下降的幅度在比值上小于某一容差限 ftol。因为 tol 通常不能小于机器精度的平方根值，所以将 ftol 置为机器精度的数量级是最合适不过的（也可以稍放大一些以免受舍入误差的影响）。

但是在因为某种原因迭代无法终止的情况下，上述终止准则中却没有一种能够“奏效”。这时我们不妨从某个已被认为是找到的极小点处开始，重新运行多维极小化程序，但种做法需要对所有辅助输入参数重新初始化，例如在下降的单纯形法中，我们需用式(10.4.1)对单纯形 $N+1$ 个顶点中的 $N$ 个进行重新设置，其中 $\mathbf{p}_0$ 为已确认的极小值顶点之一。

“重新开始”做起来不应该很费事，因为毕竟算法已经有了一次收敛的“经历”，它所收敛到的点即是“重新开始”的起点，而“重新开始”也不只过是拿起原算法“重操旧业”。

接下来，我们来看一看 $N$ 维情况下的 **amoeba** 是如何运行的：

```
#include <math.h>
#include "nrutil.h"
#define NMAX 5000           允许计算函数值的最多次数
#define GET_PSUM \
                        for (j=1;j<=ndim;j++) {\
                        for (sum=0.0,i=1;i<=mpts;i++) sum += p[i][j];\
                        psum[j]=sum;}
#define SWAP(a,b) {swap=(a);(a)=(b);(b)=swap;}

void amoeba(float **p, float y[], int ndim, float ftol, float (*funk)(float []), int *nfunk)
  用内尔德和米德提出的下降单纯形法求多元函数 funk(x)的极小值，其中 x[1..ndim]为 ndim 维向量，矩阵 p[1..
  ndim+1]为输入参数，它的 ndim+1 个 ndim 维的行向量组成初始单纯形的 ndim+1个顶点。输入项中还包括向量 y
  [1..ndim+1]及 ftol，其中 y[1..ndim+1]的每一分量都需预先初始化为函数 funk 在单纯形的 ndim+1个顶点处的
  值，ftol 须预置为收敛时函数值(注意!)的相对容差限。输出时，p 和 y 又将作为输出参数被赋值为 ndim+1个新的
  点，在这些点处函数值在极小函数值的容差限 ftol 范围内。另一输出参数 nfunk 将给出程序中函数值的求值次数。
{
    float amotry(float **p, float y[], float psum[], int ndim,
        float (*funk)(float[]), int ihi, float fac);
    int i,ihi,ilo,inhi,j,mpts=ndim+1;
    float rtol, sum, swap, ysave, ytry, *psum;

    psum=vector(1,ndim);
    *nfunk=0;
    GET_PSUM
    for (;;) {
        ilo=1;              首先确定最高点(结果最差的点)，次高点和最低点(结果最好的点)
        ihi = y[1]>y[2] ? (inhi=2,1) : (inhi=1,2);        确定的方法是对单纯形中的点作循环
        for (i=1;i<=mpts;i++) {
            if (y[i] <= y[ilo]) ilo=i;
            if (y[i] > y[ihi]) {
                inhi=ihi;
                ihi=i;
            } else if (y[i] > y[inhi] && i != ihi) inhi=i;
        }
```

```
        rtol=2.0*fabs(y[ihi]-y[ilo])/(fabs(y[ihi])+fabs(y[ilo]));      计算从最高点到最低点的范围，如果合适则返回
        if (rtol < ftol) {                                              若返回，则在第一槽内设最佳点和值
            SWAP(y[1],y[ilo])
            for (i=1;i<=ndim;i++) SWAP(p[1][i],p[ilo][i])
            break;
        }
        if (*nfunk >= NMAX) nrerror("NMAX exceeded");
        *nfunk += 2;                                  新一轮迭代的开始。首先从高点通过单纯形的表面以因子
                                                      −1作外推，即从高点将单纯形反射
        ytry=amotry(p,y,psum,ndim,funk,ihi,-1.0);
        if (ytry <= y[ilo])                             得到一个比最佳点稍好的结果，因此用因子2再作一次外推
            ytry=amotry(p,y,psum,ndim,funk,ihi,2.0);
        else if (ytry >= y[inhi]) {                  反射点如果不如次高点，取一中等程度低的点即作一次一维收缩
            ysave=y[ihi];
            ytry=amotry(p,y,psum,ndim,funk,ihi,0.5);
            if (ytry >= ysave) {                      不能去掉高点。最好围绕最低点收缩
                for (i=1;i<=mpts;i++) {
                    if (i != ilo) {
                        for (j=1;j<=ndim;j++)
                            p[i][j]=psum[j]=0.5*(p[i][j]+p[ilo][j]);
                        y[i]=(*funk)(psum);
                    }
                }
                *nfunk += ndim;                       跟踪记录函数值
                GET_PSUM                              重新计算 psum
            }
        } else --(*nfunk);                            修正估算值
    }
    free_vector(psum,1,ndim);                         返回测试处，并作下一轮迭代
}


#include "nrutil.h"

float amotry(float **p, float y[], float psum[], int ndim,
    float (*funk)(float []), int ihi, float fac)
    从高点通过单纯形的表面以因子 fac 试作外推，若新求出的点结果优于高点，则以前者代后者
{
    int j;
    float fac1,fac2,ytry,*ptry;

    ptry=vector(1,ndim);
    fac1=(1.0-fac)/ndim;
    fac2=fac1-fac;
    for (j=1;j<=ndim;j++) ptry[j]=psum[j]*fac1-p[ihi][j]*fac2;
    ytry=(*funk)(ptry);                                 计算函数在试探点处的值
    if (ytry < y[ihi]) {                                如果试探点的结果优于最高点，则以前者替代后者
        y[ihi]=ytry;
        for (j=1;j<=ndim;j++) {
            psum[j] += ptry[j]-p[ihi][j];
            p[ihi][j]=ptry[j];
        }
    }
    free_vector(ptry,1,ndim);
    return ytry;
}
```

参数文献和进一步读物：

Nelder, J.A., and Mead, R. 1965, *Computer Journal*, vol. 7, pp. 308~313. [1]

## 10.5 多维情况下的方向集(鲍威尔)方法

从10.1节到10.3节中我们已经知道如何求一元函数的极小值。若从 $N$ 维空间中的某点 **p** 出发，沿某一向量的方向 **n** 进行搜索，那么任何一个 $N$ 元函数 $f(\mathbf{p})$ 都可沿方向 **n** 用一维方法求出极小值。因此我们可以设想，各种各样的多维极小化方法，都可以用一系列这样的沿某一方向的一维方法组合而成，各种方法之间仅仅由于每一步中所选择的下一搜索方向 **n** 的不同而不同。所有这样的方法都要假定存在一个"黑箱"子算法，我们称之为"linmin"(本节最后将有详细的程序给出)，其定义如下：

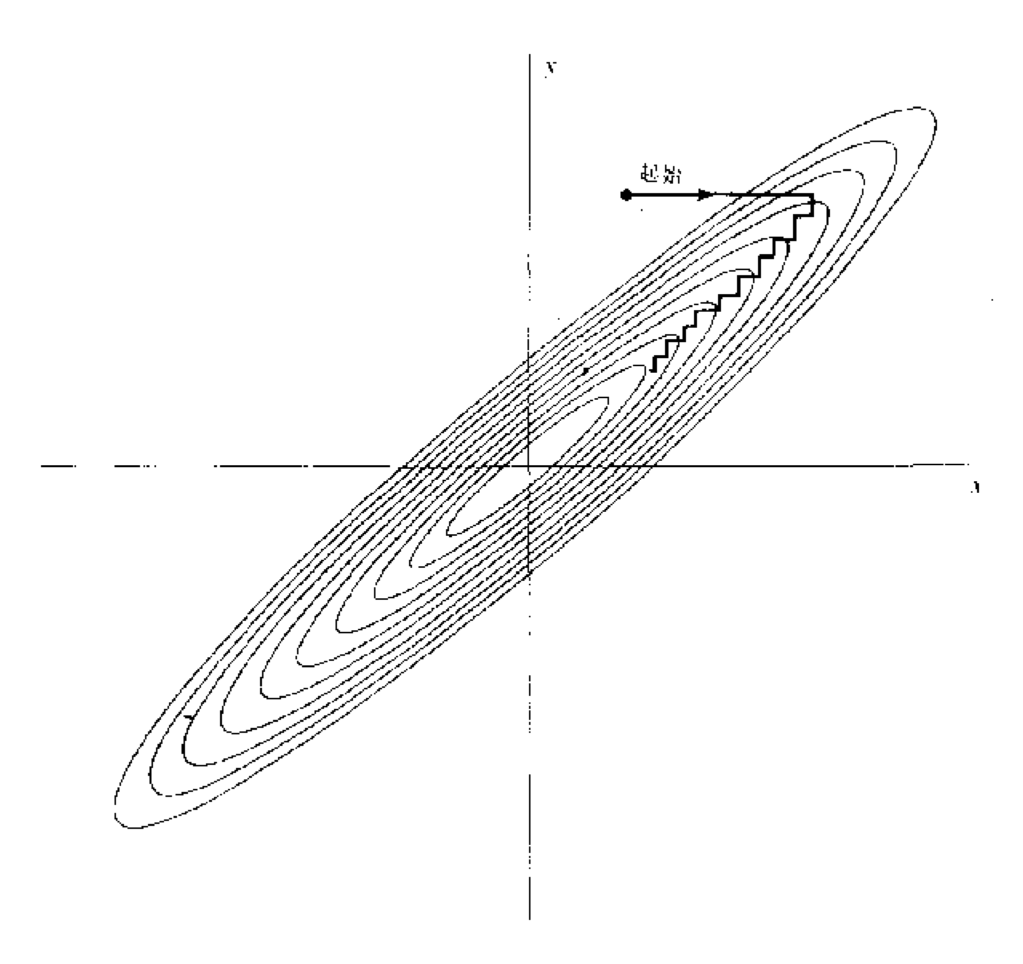
> **linmin**：给定输入参数向量 **P**,**n** 及函数 $f$，求出使 $f(\mathbf{P}+\lambda\,\mathbf{n})$ 达到最小的标量 $\lambda$；然后以 $\mathbf{P}+\lambda\,\mathbf{n}$ 替代 **P**，用 $\lambda\,\mathbf{n}$ 替代 **n**，结束。

本节及后面两节中所有的求极小值方法，都将基于这种逐步一维搜索的一般思想。(第10.7中的算法不需要精确的一维搜索，它有自己专门的近似一维搜索子程序 **lnsrch**)。本节中将讨论的方法在选择搜索方向时，不包含函数梯度。读者将注意到，在 **linmin** 中不需特别说明是否要利用梯度信息，读者可以自己选择，优劣程度由目标函数本身决定。但是大多数读者可能会倾向于在 **linmin** 中利用梯度信息，而在选择搜索方向时忽略这些信息，因为这样可以大大地减少整个过程的计算量。

但是，如果目标函数的梯度不能计算又该如何呢？读者可能首先会想到这样一种简单的方法：取单位向量 $\mathbf{e}_1,\mathbf{e}_2,\ldots,\mathbf{e}_N$ 作为一个方向集合，利用 **linmin** 算法，沿 $\mathbf{e}_1$ 搜索达到该方向上的极小值，然后再从这个极小值处开始，沿第二个方向 $\mathbf{e}_2$ 搜索，求得函数在 $\mathbf{e}_2$ 方向上的极小值，这样一直下去，根据所需绕整个方向集做尽可能多次数的循环，直到函数值不再减小。

实际上，这种策略方法对，许多函数来说应用的效果并不坏。但它对其他一些函数求解的效率就不很高，这其中的原因非常有趣。让我们来考虑，某个二元函数，其等高线图(水平线)恰好是一个与坐标轴成某一角度的狭长的山谷(见图10.5.1)。对于这种函数，在每一阶段平行于坐标轴"沿山谷的高度下降方向"搜索的路径，只能是一系列极小的步长。在更一般的 N 维情况，如果函数二阶导数的大小，在某些方向上比在其他方向上大得多，那么为收敛到某点就需要绕全部 N 个基本向量转很多圈才能达到目的。这种情况并不属罕见，你越不想碰到就越会碰到，因此我们不能回避它。

由此看来，我们显然需要比[$\mathbf{e}_i$]更好的方向集。所有的**方向集方法**都要求在运行当中能给出更新方向集的规则，并使新的方向集满足：(i)在其中某几个相当好的方向上，可使搜索路径沿窄谷前进很长一段距离；或者(ii)包含某些"互不干扰"的方向，它们具有这样一种特殊性质，即：沿某一个方向的一维搜索，与沿另一方向进行的下一轮一维搜索，不会相互干扰。这个比(i)更精细约束条件可避免绕方向集的循环无修正地进行下去。




在一狭长"山谷"中(图中以轮廓线表示)沿坐标轴方向搜索极小值。除非山谷的方位是最佳的,否则这种逐步一维搜索的方法效率将极低,需要搜索很多步方能达到极小值,而且每步的步长都很小,还要来回穿过主轴多次。

图10.5.1

### 10.5.1 共轭方向

这种"互不相关"的方向,更规范一些称**共轭方向**。这一概念值得在此做一番详细的数学上的说明。

首先,如果沿某一方向 **u** 求一函数的极小值,那么在 **u** 方向的极小值处,该函数的梯度必然与 **u** 垂直;否则,沿 **u** 方向还将会有一个非零的方向导数。

下一步,取某一特殊点 **P** 作为坐标系的原点,若坐标系的坐标以 **x** 标记,则任意函数 **f** 可用泰勒级数近似为:

$$\begin{aligned} f(\mathbf{x}) &= f(\mathbf{P}) + \sum_i \frac{\partial f}{\partial x_i} x_i + \frac{1}{2} \sum_{i,j} \frac{\partial^2 f}{\partial x_i \partial x_j} x_i x_j + \cdots \\ &\approx c - \mathbf{b} \cdot \mathbf{x} + \frac{1}{2} \mathbf{x} \cdot \mathbf{A} \cdot \mathbf{x} \end{aligned} \tag{10.5.1}$$

上式中

$$c \approx f(\mathbf{P}) \qquad \mathbf{b} \equiv -\nabla f|_{\mathbf{P}} \qquad [\mathbf{A}]_{ij} \equiv \frac{\partial^2 f}{\partial x_i \partial x_j}\bigg|_{\mathbf{P}} \tag{10.5.2}$$

我们称矩阵 **A** 为目标函数在 **P** 点的海赛(**Hessian**)矩阵,**A** 中各元素是函数在 **P** 点的二阶偏导数。

由近似式(10.5.1)容易求得 $f$ 的梯度为:

$$\nabla f = \mathbf{A} \cdot \mathbf{x} - \mathbf{b} \tag{10.5.3}$$

(上式说明,满足 $\mathbf{A}\cdot\mathbf{x}=\mathbf{b}$ 的点可使梯度变为零,即函数在该处将取得极值,在10.7节中我们还要用到这一结论。)

当搜索路径沿某一方向移动时,梯度$\nabla f$ 将会如何变化呢?很显然

$$\delta(\nabla f) = \mathbf{A} \cdot (\delta \mathbf{x}) \tag{10.5.4}$$

假设我们已沿某方向 **u** 达到某个极小值,现在打算沿一个新的方向 **v** 进行搜索,前提是沿 **v** 的移动不破坏沿 **u** 的一维搜索,这就是说梯度始终与 **u** 保持垂直,即梯度的变化与 **u** 垂直,以公式表示,就是:(由式(10.5.4)得到)。

$$0 = \mathbf{u} \cdot \delta\nabla(f) = \mathbf{u} \cdot \mathbf{A} \cdot \mathbf{v} \tag{10.5.5}$$

如果上式对两个向量 **u** 和 **v** 成立,则称它们是共轭的。若一个向量集的所有向量两两满足这一关系式,那么称这个向量集为共轭集。若沿某共轭方向集的各个方向顺序作一维搜索求极小值,则在任何一个方向上都不需作重复搜索(当然,除非破坏了共轭集的性质,即在一个与方向集不共轭的方向上作一维搜索。)

一个方向集方法成功的关键在于能够构造出一个方向集,其 $N$ 个方向线性无关、且两两共轭。那么对于形如(10.5.1)的二次式,只需沿 $N$ 个方向作一遍一维搜索,就可以精确地求出其极小值。对于不具有精确二次形式的函数,一轮迭代不能精确地求出极小值,但经过几轮一维搜索之后,也能在适当的时候二次收敛到它的极小值。

### 10.5.2 鲍威尔的二次收敛方法

鲍威尔(Powell)首先提出一种方向集方法,该方法可以构造出 $N$ 个两两共轭的方向,其构造过程如下:首先将方向集 $\mathbf{u}_i$ 初始化为坐标轴向量:

$$\mathbf{u}_i = \mathbf{e}_i \quad i = 1, \ldots, N \tag{10.5.6}$$

然后重复下列各步骤(称为"基本步骤"),直至函数值不再减小:

- 记起始位置为 $\mathbf{P}_0$
- 对 $i=1,\ldots,N$,将 $\mathbf{P}_{i-1}$ 移至目标函数沿 $\mathbf{u}_i$ 方向的极小值点,并记此点为 $\mathbf{P}_i$
- 对 $i=1,\ldots,N$,将 $\mathbf{u}_{i-1}$ 赋给 $\mathbf{u}_i$
- 置 $\mathbf{u}_N=\mathbf{P}_N-\mathbf{P}_0$
- 将 $\mathbf{P}_N$ 移至函数在 $\mathbf{u}_N$ 方向上的极小点,并记该点为 $\mathbf{P}_0$

鲍威尔曾于1964年,对于形如式(10.5.1)的二次型,上述基本步骤经过 $k$ 轮迭代以后,可以生成一个方向集 $\mathbf{u}_i$,它的最后 $k$ 个方向是两两共轭的。因此,在 $N$ 轮迭代后,总共经过 $N(N+1)$次一维搜索,便可以精确地求出一个二次型的极小值。布伦特(参见[1])以一种易于理解的形式给出了上述结论的证明。

但是,鲍威尔的二次收敛算法还存在一个问题,即在每一阶段用 $\mathbf{P}_N-\mathbf{P}_0$取代 $\mathbf{u}_1$时可能会导致各方向集"互相重叠"而变成求线性相关。一旦这样的情况发生,则只能求出全 $N$ 维

空间中一个子空间内的函数极小值，也就是说，得到的结果是错误的。因此，该算法不能用于上述情况。

解决鲍威尔算法中的线性相关问题有几种途径，其中包括：

1. 在每 $N$ 轮或 $N+1$ 轮迭代以后，将方向集 $\mathbf{u}_i$ 重新初始化为基向量 $\mathbf{e}_i$，这种方法方便实用。特别是当二次收敛性对于问题本身来说，是一个需着重考虑的因素时（即目标函数非常近二次型，而且结果的精度要求较高），我们建议读者不妨试试这种方法。

2. 布伦特曾指出，将方向集重置为任意正交矩阵的列向量同样也是很好的办法。他提出将方向集重置为已计算出来的矩阵的 $\mathbf{A}$ 的主方向（他给出了确定它的步骤），而不是放弃已建立的共轭方向上的信息。计算主方向的过程从本质上来看，就是一个奇异值分解的算法（见第2.6节）。布伦特还提出另外一些很好的算法技巧，而且他对鲍威尔算法所提出的改进也可能是迄今为止最佳的一种。具体算法及程序清单请读者参阅他本人的著作。遗憾的是，这些内容都超出了本书的范围。

3. 还可以放弃二次收敛的性质，而采用一种更有启发性的方案（由鲍威尔提出），即沿窄谷找出几个好的方向向量，以代替 $N$ 个必需共轭的方向。下面我们将要讨论的就是这种方法。（这一方法也是阿克顿在[2]中所给出的鲍威尔方法的翻版，下文中有几部分内容就是从该书中截取的。）

### 10.5.3 舍弃函数值下降最多的方向

现在我们打算放弃二次收敛性，但这是不是因为吃不着葡萄就说葡萄是酸的。或者说二次收敛性究竟是不是那么重要呢？这依赖于目标函数本身。有些目标函数呈长且迂回的山谷状。如果一个程序的运行必须象进行障碍滑雪赛一样沿着弯弯曲曲的山谷路径绕来绕去向下搜索。那么二次收敛性就显得没有什么特殊的价值了。沿着这一长方向，一种具有二次收敛性的算法总是试图外推到某条抛物线的极小值，而这条抛物线根本不能拟合出来；同时，$N-1$个横截方向的共轭性逐渐被这种曲折性破坏掉。

但是，算法或迟或早总能收敛于一个近似椭圆的极小值（参看式(10.5.1)中当梯度 $\mathbf{b}$ 为零时的情况）。那么，根据精度要求，具有二次收敛性的算法可以节省几倍 $N^2$ 的附加的一维搜索计算量，因为二次收敛性在每轮迭代中都使有效位数翻倍。

我们对鲍威尔算法所作的改进，在基本思想上仍是置 $\mathbf{P}_N-\mathbf{P}_0$ 为一新的搜索方向，因为它毕竟是经过对全部 $N$ 个可能的方向试探以后得到的平均方向。对于长方向迂回缓慢的山谷，这个方向可能是沿新长方向的好路径。与原方法比较，这里的改变之处就在于，放弃函数值下降最多的原方向，这看来似乎有些不合理，因此该方向是上一轮迭代中的最好方向。但是，这也可能是我们所添加的这个新方向的一个主要成分。因此，放弃它反而给我们提供一个绝好机会，可以避免线性相关情况的出现。

但也有几种情况例外。有时候不增加任何新方向反而好些，我们定义

$$f_0 \equiv f(\mathbf{P}_0) \quad f_N \equiv f(\mathbf{P}_N) \quad f_E \equiv f(2\mathbf{P}_N - \mathbf{P}_0) \tag{10.5.7}$$

这里 $f_E$ 为函数在某“外推”点处的值，该点沿所设置的新方向较远。同时定义 $\Delta f$ 为本轮迭代中沿某特定方向函数值的最大下降量（$\Delta f$ 为一正值），则有：

1. 若 $f_E \geqslant f_0$，则保持原方向集不变进入下一轮迭代，因为这种情况下平均方向 $\mathbf{P}_N-\mathbf{P}_0$ 已经毫无用处。

2. 若$2(f_0-2f_N+f_E)[(f_0-f_N)-\Delta f]^2 \geqslant (f_0-f_E)^2\Delta f$，则仍以原方向集进入下一轮迭代。这是因为 i)沿平均方向函数值的下降在很大成分上并不是由于某一单个方向上的下降，或者 ii)沿平均方向有一相当大的二阶导数存在，这说明我们已经很接近极小值的底部。

上面这种鲍威尔方法的程序实现将在下文中给出。在该程序中，需说明的是，$x_i$ 为一矩阵，其各列为方向集中的元素 $\mathbf{n}_i$；至于其他记号，它们对应的内容是不言自明的。

```
#include <math.h>
#include "nrutil.h"
#define ITMAX 200                              允许迭代的最大次数

void powell(float p[], float **xi, int n, float ftol, int *iter, float *fret, float (*func)(float []))
```

求 n 个变量的函数 func 的极小值。输入参数包括初始点 p[1..n]、初始矩阵 xi[1..n][1..n]（其列向量给出初始方向集（通常是 n 个单位向量））、以及函数的容差限 ftol（当某轮迭代中，函数值的下降幅度小于该值时，迭代终止）。输出时，p 被置为所求得的极小点，xi 为迭代终止时的方向集，fret 为返回的 p 点之函数值，iter 为迭代次数。在本程序中调用子程序 **linmin**。

```
{
    void linmin(float p[], float xi[], int n, float *fret,
        float (*func)(float []));
    int i,ibig,j;
    float del,fp,fptt,t,*pt,*ptt,*xit;

    pt=vector(1,n);
    ptt=vector(1,n);
    xit=vector(1,n);
    *fret=(*func)(p);
    for (j=1;j<=n;j++) pt[j]=p[j];                记录初始点
    for (*iter=1;;++(*iter)) {
        fp=(*fret);
        ibig=0;
        del=0.0;                      该值将作为函数值最大的下降量
        for (i=1;i<=n;i++) {          在每轮迭代中，对方向集中所有方向进行循环
            for (j=1;j<=n;j++) xit[j]=xi[j][i];      将该方向复制下来
            fptt=(*fret);
            linmin(p,xit,n,fret,func);               沿该方向求极小
            if (fabs(fptt-(*fret)) > del) {          如果它是迄今为止函数值下降量中最
                del=fabs(fptt-(*fret));                  大的，则将其记录下来
                ibig=i;
            }
        }
        if (2.0*fabs(fp-(*fret)) <= ftol*(fabs(fp)+fabs(*fret))) {
            free_vector(xit,1,n);                    终止准则
            free_vector(ptt,1,n);
            free_vector(pt,1,n);
            return;
        }
        if (*iter == ITMAX) nrerror("powell exceeding maximum iterations.");
        for (j=1;j<=n;j++) {                         构造外推点和移动的平均方向，将老的
            ptt[j]=2.0*p[j]-pt[j];                       起始点记录下来
            xit[j]=p[j]-pt[j];
            pt[j]=p[j];
        }
        fptt=(*func)(ptt);                           外推点处的函数值
        if (fptt < fp) {
            t=2.0*(fp-2.0*(*fret)+fptt)*SQR(fp-(*fret)-del)-del*SQR(fp-fptt);
            if (t < 0.0) {
                linmin(p,xit,n,fret,func);           找出新方向的极小点，并将这个新方
                for (j=1;j<=n;j++) {                     向记录下来
                    xi[j][ibig]=xi[j][n];
                    xi[j][n]=xit[j];
                }
```

```
            }
        }
    }                                          返回进行下的轮迭代
}
```

## 10.5.4 线性极小化的程序实现

用第10.1节—第10.3节中所述的一维搜索方法可以正确无误地实现 **linmin**，所不同的是，这里要以向量点 **P**（这些点都位于某一给定的方向 **n** 上）的形式重写那些算法，而不能再用标量 $x$。尽管这项工作很明确，但程序实现却相当繁琐，其间要有许多 "for(k=1;k<=n;k++)" 的循环。

由于篇幅所限，本书中不能给出详细的程序清单，这里的 linmin 只是一种运行得尚可的框架，它构造了一个一元"模拟"函数 **f1dim**，这个函数是目标函数 **func** 沿通过 **p** 点的直线方向 xi 上的值。**linmin** 调用我们很熟悉的一维算法 **mnbrak**（第10.1节）以及 **brent**（第10.2节）来求 f1dim 的极小值，它同 f1dim 的信息传递是通过全程变量（外部量），而"跨入"**mnbrak** 和 **brent**。这是 **linmin** 在将指针变量传递给 **f1dim** 时也采用了这种方式，其中这个指针变量指向用户提供的函数。

但 **linmin** 作为多维极小化与一维极小化程序之间的接口产生一些不必要的向量复制，这是它唯一的缺陷。一般来讲程序不会因此增加很多的计算量，但这一缺陷却是无法逃避的。

```
#include "nrutil.h"
#define TOL 2.0e-4                传送到 brent 的容差限

int ncom;                         与 f1dim 通信的全局变量
float *pcom, *xicom, (*nrfunc)(float [])

void linmin(float p[], float xi[], int n, float *fret, float(*func)(float[]))
    给定一个 n 维点 p[1..n]和一个 n 维方向 xi[1..n]，从 p 点开始沿 xi 方向移动，不断重置 p 点，直至 func(p)取得 xi
    方向上的极小值，并置 xi 为 p 所移动的实际向量位移，同时返回在还有 fret 它是函数 func 返回在 p 点的值。实际上
    该程序完全是通过调用 mnbrak 和 brent 来实现的。
{
    float brent(float ax, float bx, float cx,
        float (*f)(float), float tol, float *xmin);
    float f1dim(float x);
    void mnbrak(float *ax, float *bx, float *cx, float *fa, float *fb,
        float *fc, float (*func)(float));
    int j;
    float xx,xmin,fx,fb,fa,bx,ax;

    ncom=n;                       定义全局变量
    pcom=vector(1,n);
    xicom=vector(1,n);
    nrfunc=func;
    for (j=1;j<=n;j++) {
        pcom[j]=p[j];
        xicom[j]=xi[j];
    }
    ax=0.0;                       解区间的初始预测
    xx=1.0;
    mnbrak(&ax,&xx,&bx,&fa,&fx,&fb,f1dim);
    *fret=brent(ax,xx,bx,f1dim,TOL,&xmin);
    for (j=1;j<=n;j++) {          构造要返回的向量结果
        xi[j] *= xmin;
```

```
        p[j] += xi[j];
    }
    free_vector(xicom,1,n);
    free_vector(pcom,1,n);
}
```

```
#include "nrutil.h"

extern int ncom;                        linmin中定义
extern float *pcom,*xicom,(*nrfunc)(float []);

float f1dim(float x)
必须伴随程序 linmin.
{
    int j;
    float f,*xt;

    xt=vector(1,ncom);
    for (j=1;j<=ncom;j++) xt[j]=pcom[j]+x*xicom[j];
    f=(*nrfunc)(xt);
    free_vector(xt,1,ncom);
    return f;
}
```

参考文献和进一步读物：


Brent, R. P. 1973, *Algorithms for Minimization Without Derivatives* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 7. [1]

Acton, F. S. 1970, *Numerical Methods That Work*, 1990, corrected edition (Washington: Mathematical Association of America), pp. 464～467. [2]


## 10.6 多维共轭梯度法

现在我们来考察这样一种情形，即在某一给定的 $N$ 值维空间中的点 $\mathbf{P}$ 处，不仅函数值 $f(\mathbf{P})$ 可计算，而且其梯度（即一阶偏导数向量）$\nabla f(\mathbf{P})$ 也是可计算的。

我们只要经过一些简单的计算论证就可以发现使用梯度信息的好处所在：设 $f$ 为一近似二次型，如式(10.5.1)所示：

$$f(\mathbf{x}) \approx c - \mathbf{b} \cdot \mathbf{x} + \frac{1}{2}\mathbf{x} \cdot \mathbf{A} \cdot \mathbf{x} \qquad (10.6.1)$$

则 $f$ 中未知参数的个数与 $\mathbf{A}$, $\mathbf{b}$ 中自由参数的个数相同，都为 $\frac{1}{2}N(N+1)$ 个，具有 $N^2$ 数量级。改变其中任意一个参数，都能使极小点的位置发生变化。因此，在收集到与 $N^2$ 等量级的信息量以前，要想找出目标函数的极小值是没有什么指望的。

在第10.5节介绍的方向集方法中，我们是利用次数达 $N^2$ 数量级的独立一维搜索，来搜集到必要信息的，而每次一维搜索都需要"几次"（有时是很多很多的几次！）函数计算。因为每计算一次梯度可以得到 $N$ 个新的信息，因此如果运用得当，作独立一维搜索的次数只需 $N$ 数量级，本节和下节中的算法实际上都是如此。

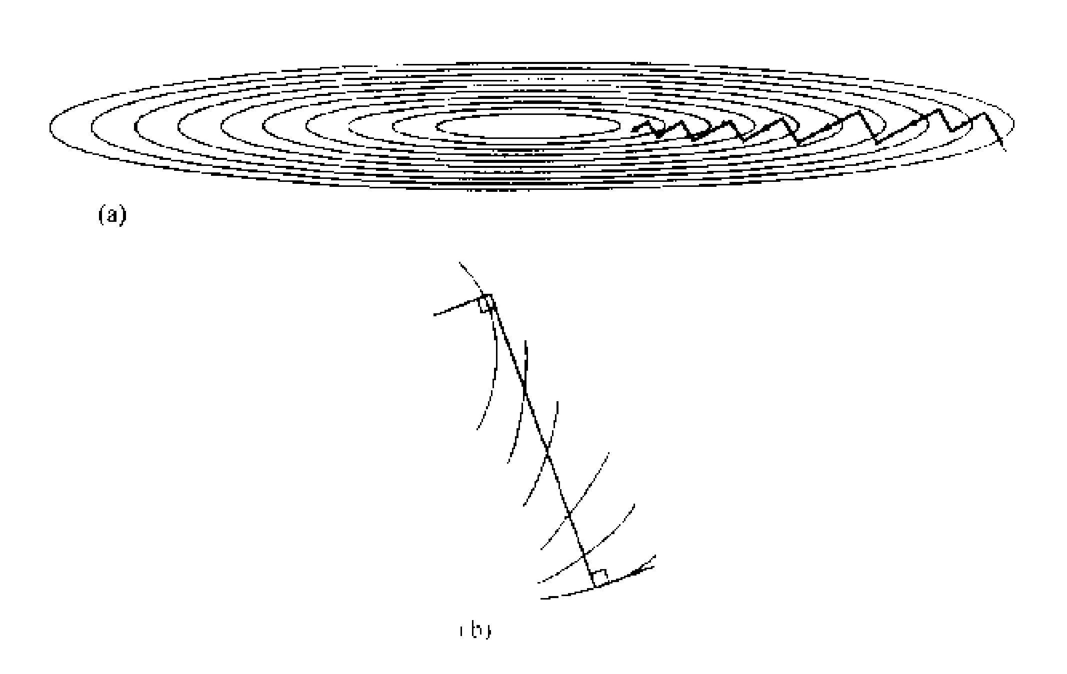
利用导数信息还可使计算速度改进 $N$ 倍，这是很明显的。粗略估算一下，可以想象，计算梯度的每一分量的时间与计算函数值本身的时间大约是一样的。因此，利用导数和不利用导数的算法都需花费 $N^2$ 数量级的计算量，与计算函数值等价。但是，即使优越性不在于数量

级为 $N$ 的计算量，它仍是相当重要的：(i)每计算一次梯度分量，一般可节约的函数值计算不只一次，而是几次，一般可以节约与一次完整的线性极化的计算等价量。(ii)在函数梯度各分量的计算公式中，常有很高阶的冗余量，如遇这种情况，特别是当函数值的计算中也有很多冗余时，则梯度的计算量能比 $N$ 次函数计算的量少得多。

初学者常犯的毛病，认为只要合理地引入梯度信息，就能获得差不多和其他算法一样好的效果。由这种思路可推出如下的并不算很好的算法，即"**最速下降法**"：

> 最速下降：以 $\mathbf{P}_0$ 为起始点，沿过 $\mathbf{P}_i$ 的路径，在局部下降梯度 $-\nabla f(\mathbf{P}_i)$ 方向上作一维极小化搜索求得 $\mathbf{P}_{i+1}$，并将 $\mathbf{P}_i$ 移至 $\mathbf{P}_{i+1}$ 点，重复这一过程直到所需的次数。

最速下降法(顺便提一句，它可以完全归结于柯西(Cauchy)算法)所遇到的麻烦与图10.5.1中所示的问题一样。即使目标函数为一严格的二次型，该方法要在沿某一狭长山谷下降的过程中，执行很多次步长很小的步骤。例如在二维场合，你可能会希望迭代第一步就到达谷底，第二步便能直接沿长轴下降，新的梯度方向总是与刚刚经过的方向垂直，因此，使用最速下降法时，必须作适当角度转动，而这种角度转动一般并不能找到极小值(见图10.6.1)。




(a)对一狭长函数使用最速下降法，虽然该方法比图10.5.1所示的方法快一些，但仍不能作为一种好的算法，在到达谷底之前也要经过很多步。(b)将最速下降法执行过程中的某一步放大来看。从局部梯度方向出发(这一方向与轮廓线的切线垂直)，经过一维搜索到达局部极小值，此时一维搜索路径与那里的轮廓线的切线平行。

图10.6.1

与推导式(10.5.5)所作的讨论一样，我们特别希望有一种办法。不是沿新的梯度方向下降，而是构造一个与原梯度方向共轭的方向，而且该方向还要尽可能地与在此以前所有已经过的方向共轭。能够实现这一构造过程的方法称为**共轭梯度方法**。

在第2.7节中我们也曾讨论过共轭梯度法，那节的内容是通过极小化二次型来求解线性代数方程；虽然共轭梯度法当时是作为一种求解技巧来介绍的，但那套公式对于本节求一个以二次型近似式((10.6.1))的函数的极小值这类问题也同样适用。我们以任一向量 $\mathbf{g}_0$ 为起点，令 $\mathbf{h}_0=\mathbf{g}_0$，并用下面的递推公式来构造两个向量序列：

$$\mathbf{g}_{i+1}=\mathbf{g}_i-\lambda_i\mathbf{A}\cdot\mathbf{h}_i \qquad \mathbf{h}_{i+1}=\mathbf{g}_{i+1}-\gamma_i\mathbf{h}_i \qquad i=0,1,2,\ldots \tag{10.6.2}$$

按这种方式构造出来的向量可同时满足正交性和共轭性：

$$\mathbf{g}_i\cdot\mathbf{g}_j=0 \qquad \mathbf{h}_i\cdot\mathbf{A}\cdot\mathbf{h}_j=0 \qquad \mathbf{g}_i\cdot\mathbf{h}_j=0 \qquad i\neq j \tag{10.6.3}$$

其中(10.6.2)中标量 $\lambda_i$ 和 $\gamma_i$ 的计算公式是：

$$\lambda_i=\frac{\mathbf{g}_i\cdot\mathbf{g}_i}{\mathbf{h}_i\cdot\mathbf{A}\cdot\mathbf{h}_i}=\frac{\mathbf{g}_i\cdot\mathbf{h}_i}{\mathbf{h}_i\cdot\mathbf{A}\cdot\mathbf{h}_i} \tag{10.6.4}$$

$$\gamma_i=\frac{\mathbf{g}_{i+1}\cdot\mathbf{g}_{i+1}}{\mathbf{g}_i\cdot\mathbf{g}_i} \tag{10.6.5}$$

(10.6.2)～(10.6.5)这组公式与(2.7.3)～(2.7.35)的不同之处除了在前者中 $\mathbf{A}$ 是对称的之外，其他仅仅是记号上的不同。(针对求函数极小值问题推导这些结论的全部过程在波拉克(Polak)[1]的书中有详细介绍。)

现在我们假设式(10.6.1)中的海赛(Hessian)矩阵 $\mathbf{A}$ 是已知的，那么我们就可以利用(10.6.2)来构造一系列的共轭方向 $\mathbf{h}_i$，并沿这些方向进行一维搜索。这样作 $N$ 次以后，就能求出二次型的极小值。但是，事实上 $\mathbf{A}$ 是未知的。

下面我们给出一个著名的定理来解决上面的问题。假设在某点 $\mathbf{P}_i$ 恰有 $\mathbf{g}_i=-\nabla f(\mathbf{P}_i)$，其中 $f$ 如(10.6.1)式所示。再假设从 $\mathbf{P}_i$ 开始沿 $\mathbf{h}_i$ 方向到达 $f$ 的某个局部极小值点 $\mathbf{P}_{i+1}$，并置 $\mathbf{g}_{i+1}=-\nabla f(\mathbf{P}_{i+1})$，那么这个 $\mathbf{g}_{i+1}$ 应该与由(10.6.2)式构造的向量完全相同。(至此，我们并没有利用 $\mathbf{A}$ 构造出 $\mathbf{g}_i$！)

证明：由式(10.5.3)得到 $\mathbf{g}_i=-\mathbf{A}\cdot\mathbf{P}_i+\mathbf{b}$ 及

$$\mathbf{g}_{i+1}=-\mathbf{A}\cdot(\mathbf{P}_i-\lambda\mathbf{h}_i)+\mathbf{b}=\mathbf{g}_i-\lambda\mathbf{A}\cdot\mathbf{h}_i \tag{10.6.6}$$

其中 $\lambda$ 的取值及 $f$ 达到该方向的极小值。但在这个极小值点，$\mathbf{h}_i\cdot\nabla f=-\mathbf{h}_i\cdot\mathbf{g}_{i+1}=0$。将后面的这个等式与式(10.6.6)联立可以很容易求 $\lambda$，而结果恰好是式(10.6.4)所示的表达式。再以这个 $\lambda$ 值代入式(10.6.6)中，所得表达式(10.6.2)完全相同。证毕。

这样，对于既不需要已知赫斯恩矩阵 $\mathbf{A}$，甚至也不需对其花费存贮空间的算法，我们已找到了它的理论基础。只需利用一维搜索和梯度向量的计算，并设置一个辅助向量存放序列 $\mathbf{g}$ 的当前向量，方向序列 $\mathbf{h}_i$ 就可以被构造出来。

以上我们所讨论的算法是共轭梯度法的弗莱彻－里弗斯(Fletcher-Reeves)版本的原

形。后来波拉克(Polak)和里拜尔(Ribiere)对这一版本做了一处小小的改动,虽然改动不大,但有些情况下其意义却非同一般。他们建议用下述表达式

$$\gamma_i = \frac{(\mathbf{g}_{i+1} - \mathbf{g}_i) \cdot \mathbf{g}_{i+1}}{\mathbf{g}_i \cdot \mathbf{g}_i} \tag{10.6.7}$$

来替代式(10.6.5)。介绍到这里,读者肯定会问,根据正交条件(10.6.3),这两个表达式不是等价的吗?不错,对精确二次型它们是等价的,但在实际当中,目标函数常常不是精确的二次型,因此在求出假设的极小值后,还需另作一番迭代。实践证明,波拉克和里拜尔提出的方法恰恰可以非常漂亮地实现向深入迭代的过度:一旦算法偏离路径,则重置 **h** 为沿局部梯度下降的方向,这就相当于重新开始执行轭梯度法。

下面这段程序是波拉克-里拜尔方法的一个变异版本,我们只要改变程序的某一局部,它就变成了弗莱彻-里弗斯方法。该程序中利用了函数 **func**(p),(其中 p[1..n]是一个 $n$ 维向量)和函数 **dfunc**(p,df),其中 **dfunc**(p,df)用来计算输入点 p 处的向量梯度 df[1..n]。

程序中还调有了 **linmin** 来执行一维搜索。正如和以前讨论过的一样,我们须把 **linmin** 中的 **brent** 改为 **dbrent**,也就是说在一维搜索中需要利用梯度信息。请读者参下面的注释说明。

```
#include <math.h>
#include "nrutil.h"
#define ITMAX 200              允许迭代的最大次数
#define EPS 1.0e-10            对收敛到零函数值的特殊情况进行调整的最小数
#define FREEALL free_vector(xi,1,n);free_vector(h,1,n);free_vector(g,1,n);

void frprmn(float p[], int n, float ftol, int *iter, float *fret, float (*func)(float []),
    void (*dfunc)(float [], float []))
    给定起始点p[1..n],利用子程序dfunc提供的梯度信息,对函数func用弗莱彻-里弗斯-波拉克里尔方法求极小值。
    函数值的收敛容差限以输入参数ftol给出。返回项包括p(极小值点的坐标)、iter(总共执行的迭代次数)以及fret
    (函数的极小值),本程序还调用了linmin执行一维搜索.
{
    void linmin(float p[], float xi[], int n, float *fret,
        float (*func)(float []));
    int j,its;
    float gg,gam,fp,dgg;
    float *g,*h,*xi;

    g=vector(1,n);
    h=vector(1,n);
    xi=vector(1,n);
    fp=(*func)(p);                              初始化
    (*dfunc)(p,xi);
    for (j=1;j<=n;j++) {
        g[j] = -xi[j];
        xi[j]=h[j]=g[j];
    }
    for (its=1;its<=ITMAX;its++) {              迭代循环
        *iter=its;
        linmin(p,xi,n,fret,func);               正常返回
        if (2.0*fabs(*fret-fp) <= ftol*(fabs(*fret)+fabs(fp)+EPS)) {
            FREEALL
            return;
        }
```

```
        fp=(*func)(p);
        (*dfunc)(p,xi);
        dgg=gg=0.0;
        for (j=1;j<=n;j++) {
            gg += g[j]*g[j];
/*          dgg += xi[j]*xi[j];     */          弗莱彻-里弗斯方法
            dgg += (xi[j]+g[j])*xi[j];          波拉克-里拜尔方法
        }
        if (gg == 0.0) {                        不太可能出现的情况：如果梯度恰为0，则
            FREEALL                                 可以结束返回
            return;
        }
        gam=dgg/gg;
        for (j=1;j<=n;j++) {
            g[j] = -xi[j];
            xi[j]=h[j]=g[j]+gam*h[j];
        }
    }
    nrerror("Too many iterations in frprmn");
}
```

## 10.6.1 有关利用导数的线性极小化之说明

请重新阅读一下本章第10.5节最后一部分的内容，下面我们所要做的事情与它完全相同，只是在执行线性极小化过程中，利用了导数信息。

我们把修改后的 **linmin** 取名为 **dlinmin**；在 **dlinmin** 中所调用的子程序 **df1dim** 将紧随其后给出。

```
#include "nrutil.h"
#define TOL 2.0e-4                     这是传送到 dbrent 的容差限

int ncom;                              与 df1dim 全局变量通讯
float *pcom, *xicom,(*nrfunc)(float []);
void (*nrdfun)(float [], float []);

void dlinmin(float p[], float xi[], int n, float *fret,float (*func)(float[]),
     void (*dfunc)(float [],float []))
     给定一n维点p[1..n]和一n维方向xi[1..n]，沿xi方向不断移动p点，直至函数func(p)取得xi方向上的极小值。
     返回时置p为极小值点，并置xi为p所移动的实际向量位移。同时返回的还有fret，它是函数func在程序返回的p
     点所取的值。实际上该程序完全是通过调用mnbrak和dbrent来实现的。
{
    float dbrent(float ax, float bx, float cx,
        float (*f)(float), float (*df)(float), float tol, float *xmin);
    float f1dim(float x);
    float df1dim(float x);
    void mnbrak(float *ax, float *bx, float *cx, float *fa, float *fb,
        float *fc, float (*func)(float));
    int j;
    float xx,xmin,fx,fb,fa,bx,ax;

    ncom=n;                        定义全局变量
    pcom=vector(1,n);
    xicom=vector(1,n);
    nrfunc=func;
```

```
    nrdfun=dfunc;
    for (j=1;j<=n;j++) {
        pcom[j]=p[j];
        xicom[j]=xi[j];
    }
    ax=0.0;                              划界的初始预测
    xx=1.0;
    mnbrak(&ax,&xx,&bx,&fa,&fx,&fb,f1dim);
    *fret=dbrent(ax,xx,bx,f1dim,df1dim,TOL,&xmin);
    for (j=1;j<=n;j++) {                 构造要返回的向量结果
        xi[j] *= xmin;
        p[j] += xi[j];
    }
    free_vector(xicom,1,n);
    free_vector(pcom,1,n);
}


#include "nrutil.h"

extern int ncom;                              定义在 dlinmin 中
extern float *pcom,*xicom,(*nrfunc)(float []);
extern void (*nrdfun)(float [], float []);

float df1dim(float x)
{
    int j;
    float df1=0.0;
    float *xt,*df;

    xt=vector(1,ncom);
    df=vector(1,ncom);
    for (j=1;j<=ncom;j++) xt[j]=pcom[j]+x*xicom[j];
    (*nrdfun)(xt,df);
    for (j=1;j<=ncom;j++) df1 += df[j]*xicom[j];
    free_vector(df,1,ncom);
    free_vector(xt,1,ncom);
    return df1;
}
```

参考文献和进一步读物：


Polak, E. 1971, *Computational Methods in Optimization*(New York: Academic Press), § 2.3. [1]

Jacobs, D. A. H. (ed) 1977, *The State of the Art in Numerical Analysis*(London: Academic Press), Chapter III.1.7 (by K. W. Brodlie). [2]


## 10.7 多维变度量法

**变度量法**(有时也称为**拟牛顿法**)的目标是从逐步的线性极小化中不断积累信息，使得算法在经过 $N$ 次线性极小化以后，便可精确地收敛于一个 N 元二次型的极小值，这与前面讨论的共轭梯度法是完全相同的。由此看来，对于更一般的光滑函数，变度量法也能达到二次收敛。

无论是变度量还是共轭梯度法都要求能计算出目标函数在任意点的梯度(即一阶导数向量)。与共轭梯度法所不同的是，变度量法需要不断存贮和更新已有的信息。这种方法所需的不是数量级为 N(这里 $N$ 为维数)的中间存储量，而是一个 $N\times N$ 大小的矩阵。一般情况下，只要 $N$ 不太大，这样大小的矩阵是根本不成问题的。

另一方面，根据我们所掌握的情况，除了历史原因以外，变度量法与共轭梯度法相比较并不拥有任何压倒优势。由于变度量法建立得较早而且推广得较为广泛，因此它现在已经能满足众多使用者的要求。除此之外，一些用变度量法开发出来的优秀软件实施(这些内容超出了本书的范围。见下)被用于解决舍入误差的极小化、特殊条件的处理等高难度问题。虽然本书倾向于使用变度量法，而不是共轭梯度法，但我们没有理由把这一习惯强加给读者。

变度量法主要分为两大类，一类是**戴维登**(Davidon)-**弗莱彻**(Fletcher)-**鲍威尔**(Powell)(即DFP)算法(有时简称Fletcher-Powell算法)；另一类是**布罗依登**(Broyden)-**弗莱彻-戈德法伯**(Goldfarb)-**香农**(Shanno)算法(即BFGS)。

BFGS算法与DFP算法的不同之处仅仅在于舍入误差、收敛容差限以及一些类似的"恼人"的细节的问题上。它们都超出了本书所讨论的范围，参阅[1,2]。但是，人们已经普遍认识到，从经验上来看，BFGS方法在这些枝节问题上比DFP方法要技高一筹，因此本节中我们将着重研究BFGS方法。

和前面一样，我们假设任意函数$f(\mathbf{x})$可用(10.6.1)式那样的二次型作局部的近似，但是对二次型的参数$\mathbf{A}$,$\mathbf{b}$的值仍一无所知，我们仅仅只能从函数值的计算和线性极小化中找到一些有关的信息。

变度量法的基本思想就是，用逐步迭代的方式求出逆海赛(Hesse)矩阵$\mathbf{A}^{-1}$的一个足够好的近似矩阵，即构造出一个矩阵序列$\mathbf{H}_i$，使其具有如下性质：

$$\lim_{i\to\infty}\mathbf{H}_i=\mathbf{A}^{-1} \tag{10.7.1}$$

如果这个极限值经过$N$次而不是$\infty$次迭代以后就能到，则是更好不过的了。

现在，我们来解释一下为什么变度量法有时也被称作"拟牛顿法"。考虑一种求函数极小值的方法，即用牛顿法搜索该函数导数的零点。在当前点$\mathbf{x}_i$的某邻域附近，我们有：

$$f(\mathbf{x})=f(\mathbf{x}_i)+(\mathbf{x}-\mathbf{x}_i)\cdot\nabla f(\mathbf{x}_i)+\frac{1}{2}(\mathbf{x}-\mathbf{x}_i)\cdot\mathbf{A}\cdot(\mathbf{x}-\mathbf{x}_i) \tag{10.7.2}$$

进而有下式成立：

$$\nabla f(\mathbf{x})=\nabla f(\mathbf{x}_i)+\mathbf{A}\cdot(\mathbf{x}-\mathbf{x}_i) \tag{10.7.3}$$

在牛顿法中我们是通过置$\nabla f(\mathbf{x})=0$来确定下一个迭代点的，即：

$$\mathbf{x}-\mathbf{x}_i=-\mathbf{A}^{-1}\cdot\nabla f(\mathbf{x}_i) \tag{10.7.4}$$

上式中左端项是在达到精确极小以前需要经过的有限步长；而右端项在我们求出精确的$\mathbf{H}$($\approx\mathbf{A}^{-1}$)之后便可成为已知项。

我们所以在"牛顿法"之前冠以"拟"字，是因为算法中使用的不是函数$f$的真正的海赛矩阵，而是该矩阵的当前近似矩阵。这样做的效果常常比利用真正的海赛矩阵效果要好。这个似是而非的结论并不难理解。让我们考虑一下函数$f$在$\mathbf{x}_i$处的下降方向，沿这些方向$\mathbf{p}$，函数$f$的值是减小的，即$\nabla f\cdot\mathbf{p}<0$。为了让牛顿方向(10.7.4)成为下降方向，必须有满足：

$$\nabla f(\mathbf{x}_i)\cdot(\mathbf{x}-\mathbf{x}_i)=-(\mathbf{x}-\mathbf{x}_i)\cdot\mathbf{A}\cdot(\mathbf{x}-\mathbf{x}_i)<0 \tag{10.7.5}$$

也就是说$\mathbf{A}$必须是正定矩阵。一般说来，在远离极小值处，我们并不能保证海赛矩阵是正定

的。对真正的海赛矩阵，用实际的牛顿步骤只能找到函数值增加的点。拟牛顿法的基本思想是，先找一个正定对称的矩阵作为 **A** 的初始近似矩阵，然后从这个近似矩阵出发，构造一个矩阵序列 $\{\mathbf{H}_i\}$，构造的原则是保证每个 $\mathbf{H}_i$ 都是正定的和对称的。在离极小值较远处，这种构造方法能够保证算法沿下降方向进行迭代；而在接近极小值的地方，更新后的公式渐近真正的海赛矩阵这样。这样我们可以获得牛顿法所具备的二次收敛性。

当我们没有充分接近极小值的时候，即使是对正定矩阵 **A** 采用完全的牛顿步骤 **p** 也不一定能使函数值下降；原因是我们移动的距离可能过远，因而使二次近似变得毫无意义。唯一能让我们心中有数的是，在沿牛顿方向移动过程中，函数 $f$ 最初是下降的。这种情况下，第9.7节中的划界策略又可以派上用场，我们可以利用它沿牛顿步骤 **p** 的方向选择一轮迭代步骤，但我们并不能总这样做。

本书将省略 DFP 算法的严格推导，有兴趣的读者请参考[3]中的详细推导过程。根据 Brodlie([2])，我们将在下文中给出该方法的启发式机理。

以 $\mathbf{x}_{i+1}$ 替代方程(10.7.4)中的 $\mathbf{x}_i$，再用所得方程减去原来的式(10.7.4)我们有：

$$\mathbf{x}_{i+1} - \mathbf{x}_i = \mathbf{A}^{-1} \cdot (\nabla f_{i+1} - \nabla f_i) \tag{10.7.6}$$

其中 $\nabla f_j = \nabla f(\mathbf{x}_j)$。从 $\mathbf{x}_i$ 得到 $\mathbf{x}_{i+1}$ 之后，我们很自然地希望新求出的估计值 $\mathbf{H}_{i+1}$ 能够象 $\mathbf{A}^{-1}$ 一样满足式(10.7.6)，即：

$$\mathbf{x}_{i+1} - \mathbf{x}_i = \mathbf{H}_{i+1} \cdot (\nabla f_{i+1} - \nabla f_i) \tag{10.7.7}$$

不仅如此，它的修正公式还应具有"$\mathbf{H}_{i+1} = \mathbf{H}_i +$修正值"这种形式。

它周围有什么"对象"可以用来构造修正项呢？最明显的是 $\mathbf{x}_{i+1} - \mathbf{x}_i$ 和 $\nabla f_{i+1} - \nabla f_i$ 这两个向量，外加矩阵序列 $\mathbf{H}_i$。有这些项来构造矩阵不可能有无穷多种方式，特别是在还要满足式(10.7.7)的情况下。

DFP 方法的修正公式给出了其中一种构造方法，它具有如下形式：

$$\begin{aligned}\mathbf{H}_{i+1} = \mathbf{H}_i &+ \frac{(\mathbf{x}_{i+1} - \mathbf{x}_i) \otimes (\mathbf{x}_{i+1} - \mathbf{x}_i)}{(\mathbf{x}_{i+1} - \mathbf{x}_i) \cdot (\nabla f_{i+1} - \nabla f_i)} \\ &- \frac{[\mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)] \otimes [\mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)]}{(\nabla f_{i+1} - \nabla f_i) \cdot \mathbf{H}_i (\nabla f_{i+1} - \nabla f_i)}\end{aligned} \tag{10.7.8}$$

上式中 $\otimes$ 表示两个向量的"外"积或"直接"积，其结果为一矩阵，$\mathbf{u} \otimes \mathbf{v}$ 的第 $i$ 行第 $j$ 列元素为 $u_i v_j$（读者可以证明式(10.7.8)确实满足式(10.7.7)）。

BFGS 的修正公式与 DFP 值的形式相同，只是多增加了一项

$$\cdots + [(\nabla f_{i+1} - \nabla f_i) \cdot \mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)]\, \mathbf{u} \otimes \mathbf{u} \tag{10.7.9}$$

其中 **u** 定义为如下向量

$$\begin{aligned}\mathbf{u} \equiv &\frac{(\mathbf{x}_{i+1} - \mathbf{x}_i)}{(\mathbf{x}_{i+1} - \mathbf{x}_i) \cdot (\nabla f_{i+1} - \nabla f_i)} \\ &- \frac{\mathbf{H}_i (\nabla f_{i+1} - \nabla f_i)}{(\nabla f_{i+1} - \nabla f_i) \cdot \mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)}\end{aligned} \tag{10.7.10}$$

(同样可以证明式(10.7.10)也满足式(10.7.7))。

只要 $f$ 是一个二次型，无论在(10.7.8)中加或不加(10.7.9)这一项，我们都能得到"$\{\mathbf{H}_i\}$在 $N$ 步之内必收敛于 $\mathbf{A}^{-1}$"的结论。对于这个结论，读者可以尽管放心。

下面的这段源代码 **dfpmin** 是拟牛顿法的程序实现，**dfpmin** 中需调用第9.7节中的 **lnsrch**。在第9.7节中 **newt** 的结尾处我们曾经提到，如果变量的比例选取不当有可能导致算法失败，下面的程序也是如此。

```
#include <math.h>
#include "nrutil.h"
#define ITMAX 200                        允许迭代的最大次数
#define EPS 3.0e-8                       机器精度
#define TOLX (4 * EPS)                   x 值的收敛准则
#define STPMX 100.0                      线性搜索中所允许的最大步长
#define FREEALL free_vector(xi,1,n);free_vector(pnew,1,n);\
free_matrix(hessin,1,n,1,n);free_vector(hdg,1,n);free_vector(g,1,n);\
free_vector(dg,1,n);

void dfpmin(float p[], int n, float gtol, int *iter, float *fret, float(*func)(float []),
    void (*dfunc)(float [], float []))
```

给定起始点 p[1..n](n 维向量)，利用子程序 **dfunc** 提供的梯度信息，用修改后的戴维登－弗莱彻－鲍威尔方法(DFP)，即布罗依登－弗莱彻－戈德法伯－香农(BFGS)方法对函数 func 求极小值。梯度零点的收敛容差限以输入参数 gtol 给出。程序的返回值包括 p[1..n](极小值点的位置坐标)、iter(总共执行的迭代次数)以及 fret(函数的极小值)。本程序中调用了子程序 **lnsrch** 执行近似线性极小化。

```
{
    void lnsrch(int n, float xold[], float fold, float g[], float p[], float x[],
        float *f, float stpmax, int *check, float (*func)(float []));
    int check,i,its,j;
    float den,fac,fad,fae,fp,stpmax,sum=0.0,sumdg,sumxi,temp,test;
    float *dg,*g,*hdg,**hessin,*pnew,*xi;

    dg=vector(1,n);
    g=vector(1,n);
    hdg=vector(1,n);
    hessin=matrix(1,n,1,n);
    pnew=vector(1,n);
    xi=vector(1,n);
    fp=(*func)(p);                          计算起始点函数值和梯度
    (*dfunc)(p,g);
    for (i=1;i<=n;i++) {                    并将逆海赛矩阵初始化为单位矩阵
        for (j=1;j<=n;j++) hessin[i][j]=0.0;
        hessin[i][i]=1.0;
        xi[i] = -g[i];                      初始化一维搜索方向
        sum += p[i]*p[i];
    }
    stpmax=STPMX*FMAX(sqrt(sum),(float)n);
    for (its=1;its<=ITMAX;its++) {          迭代循环
        *iter=its;
        lnsrch(n,p,fp,g,xi,pnew,fret,stpmax,&check,func);
        在 lnsrch中计算新的函数值并存在fp中以备下一轮线性搜索时使用

        fp = *fret;
        for (i=1;i<=n;i++) {
            xi[i]=pnew[i]-p[i];             更新搜索方向和当前点
            p[i]=pnew[i];
```

```
        }
        test=0.0;                                    检验Δx的收敛性
        for (i=1;i<=n;i++) {
            temp=fabs(xi[i])/FMAX(fabs(p[i]),1.0);
            if (temp > test) test=temp;
        }
        if (test < TOLX) {
            FREEALL
            return;
        }
        for (i=1;i<=n;i++) dg[i]=g[i];               将原梯度存放起来
        (*dfunc)(p,g);                               计算新的梯度
        test=0.0;                                    检验零梯度的收敛性
        den=FMAX(*fret,1.0);
        for (i=1;i<=n;i++) {
            temp=fabs(g[i])*FMAX(fabs(p[i]),1.0)/den;
            if (temp > test) test=temp;
        }
        if (test < gtol) {
            FREEALL
            return;
        }
        for (i=1;i<=n;i++) dg[i]=g[i]-dg[i]; 计算梯度的差值
        for (i=1;i<=n;i++) {                         并用该差值乘以当前矩阵
            hdg[i]=0.0;
            for (j=1;j<=n;j++) hdg[i] += hessin[i][j]*dg[j];
        }
        fac=fae=sumdg=sumxi=0.0;                     计算分母的点数
        for (i=1;i<=n;i++) {
            fac += dg[i]*xi[i];
            fae += dg[i]*hdg[i];
            sumdg += SQR(dg[i]);
            sumxi += SQR(xi[i]);
        }
        if (fac*fac > EPS*sumdg*sumxi) {             若fac没有足够的正数特性，这一步跳过
            fac=1.0/fac;                                 不执行
            fad=1.0/fae;
            BFGS比DFP多出的那一项:
            for (i=1;i<=n;i++) dg[i]=fac*xi[i]-fad*hdg[i];
            for (i=1;i<=n;i++) {                                BFGS修正公式
                for (j=i;j<=n;j++) {
                    hessin[i][j] += fac*xi[i]*xi[j]
                    -fad*hdg[i]*hdg[j]+fae*dg[i]*dg[j];
                }
            }
        }
        for (i=1;i<=n;i++) {                         计算下一搜索方向
            xi[i]=0.0;
            for (j=1;j<=n;j++) xi[i] -= hessin[i][j]*g[j];
        }
    }                                                并返回执行下一轮迭代
    nrerror("too many iterations in dfpmin");
    FREEALL
}
```

这里说明一点，由 **dfpmin** 实现的拟牛顿法只需近似的线性极小化（由子程序 **lnsrch** 执行）就能得到很好的结论；而第10.5节中的 **powell** 以及第10.6节中的 **frprmn** 则需要比较精确的 **linmin** 来执行线性最小化。

## 10.7.1 变度量法的进一步实现

由于舍入误差的影响有时会导致矩阵 $H_i$ 接近奇异或非正定，这种情况虽属少见，但却无法避免其发生的可能。这时，我们所推测的搜索方向可能并不是函数值下降的方向，而且当前 $H_i$ 的近似奇异性也会

“移传”给定它后面的 $\mathbf{H}_i$，因此其后果将是很严重的。

这类问题的补救办法很非常简单，我们在第10.4节也曾提到过：一旦遇到上述情况发生的倾向，应从已求出的极小点处重新开始执行该算法，看看极小点会不会跑到别处去。这种处理方法虽然简单易行，但却不够精细。现代变度量法在处理这一问题的方式上要比它复杂得多。

让我们从另一角来分析算法。如果不去构造 $\mathbf{A}^{-1}$ 的近似矩阵，而去构造近似矩阵 $\mathbf{A}$ 本身，似乎也不是没有可能，而且这样一来，我们不需直接计算(10.7.4)的左边项，只要求解如下的线性方程组

$$\mathbf{A}\cdot(\mathbf{x}_m-\mathbf{x}_i)=-\nabla f(\mathbf{x}_i) \tag{10.7.11}$$

即可。乍一看，这种想法很糟糕，因为求解(10.7.11)的计算量已达到 $N^3$ 数量级，而且也看不出它对舍入误差问题的解决究竟有怎样的帮助。其实其中的技巧在于，我们存储的是矩阵 $\mathbf{A}$ 三角分解(即乔莱斯基(Cholesky)分解，参见第2.9节)后的矩阵，而不是存储 $\mathbf{A}$ 本身。$\mathbf{A}$ 的乔莱基分解修正公式是 $N^2$ 数量级，即使在有限舍入误差存在的情况下，它也能确保矩阵正定性和非退化性质。该修正公式是由吉尔(Gill)和默里(Murray)提出的，有兴趣的读者请查阅参考文献[1,2]。

参考文献和进一步读物：

Dennis, J. E., and Schnabel, R. B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*(Englewood Cliffs, NJ: Prentice-Hall). [1]

Jacobs, D. A. H. (ed) 1977, *The State of the Art in Numerical Analysis*(London: Academic Press), Chapter III. 1, §§3~6 (by K. W. Brodlie). [2]

Polak, E. 1971, *Computational Methods in Optimization*(New York: Academic Press), pp. 56ff. [3]

## 10.8 线性规划和单纯形法

**线性规划**(有时也称为**线性优化**)所讨论的问题可归纳为如下形式：对 $N$ 个独立变元 $x_1,\ldots,x_N$，求函数

$$z=a_{01}x_1+a_{02}x_2+\ldots+a_{0N}x_N \tag{10.8.1}$$

的极大值。条件是满足 $N$ 个基本约束条件。

$$x_1\geq 0,\quad x_2\geq 0,\quad \cdots\quad x_N\geq 0 \tag{10.8.2}$$

及 $M=m_1+m_2+m_3$ 个附加约束条件。在附加约束中，有 $m_1$ 个具有形式

$$a_{i1}x_1+a_{i2}x_2+\ldots+a_{iN}x_N\leq b_i\quad(b_i\geq 0)\quad i=1,\ldots,m_1 \tag{10.8.3}$$

$m_2$ 个具有形式

$$a_{j1}x_1+a_{j2}x_2+\ldots+a_{jN}x_N\geq b_j\geq 0\quad j=m_1+1,\ldots,m_1+m_2 \tag{10.8.4}$$

其余 $m_3$ 个具有形式

$$a_{k1}x_1+a_{k2}x_2+\ldots+a_{kN}x_N=b_k\geq 0\qquad k=m_1+m_2+1,\ldots,m_1+m_2+m_3 \tag{10.8.5}$$

上面各式中，参数 $a_{ij}$ 可正可负，也可以是零；而所有参数 $b$ 则规定为非负数(如上述三式中最末尾的不等式所示)，但这只是一种惯例而已，因为我们可以利用 $-1$ 去乘任何一个不等式

的两端。此外，线性规划问题对约束条件的个数 $M$ 没有特殊限制，无论它是小于、等于还是大于变量个数 $N$ 都无关紧要。

满足约束条件(10.8.2)～(10.8.5)的所有 $x_1,\ldots,x_N$ 值组成的集合称为可行向量；欲求极大值的函数称为目标函数；使目标函数取得极大值的可行向量为最优可行向量。但是，有些情况下最优可行向量可能不存在，原因有两个：(i)根本没有可行向量存在，即给定的约束条件之间是相互排斥的；(ii)目标函数没有极大值，也就是说在 $N$ 维空间中的某个方向上，一个或多个变量可无限增大而约束条件仍旧能被满足，此时目标函数的值是无界的。

如大家所见，线性规划的研究对象中包含了许许多多的记号和术语，它们都是由一些有志于该领域研究的志同道合者杜撰出来的。实际上，线性规划中的基本思想相当简单。为简明起见，我们先通过几个特殊的例子来证明线性规划的基本问题，然后再将它们推广至一般。

为什么说线性规划如此重要呢？首先，因为"非负性"是对某类变量 $x_i$ 的一种非常普遍的约束。这些 $x_i$ 代表的是一些实际物品的明确数量，如枪炮数目、黄油量、美元数以及维生素 $E$、食物卡路里、千瓦小时、质量的单位等等，这种"非负性"得到的便是约束条件(10.8.2)。第二，因为人们常常对一些由人类或自然界施加的加法(线性)限制或界限感兴趣，如最低营养需求、最大可支付开销、现有人力或资金的最高使用限度、选民赞成票容忍达到的最低百分比等等，这些都可以抽象成为式(10.8.3)～(10.8.5)的形式。第三，由于目标函数极有可能是线性的，或至少可有线性函数来近似(因为这才是线性规划可以解决的问题)，我们才有如(10.8.1)形式的目标函数。总之，有关线性规划应用情况的综述，读者可以参考布兰德(Bland)的著作[1]。

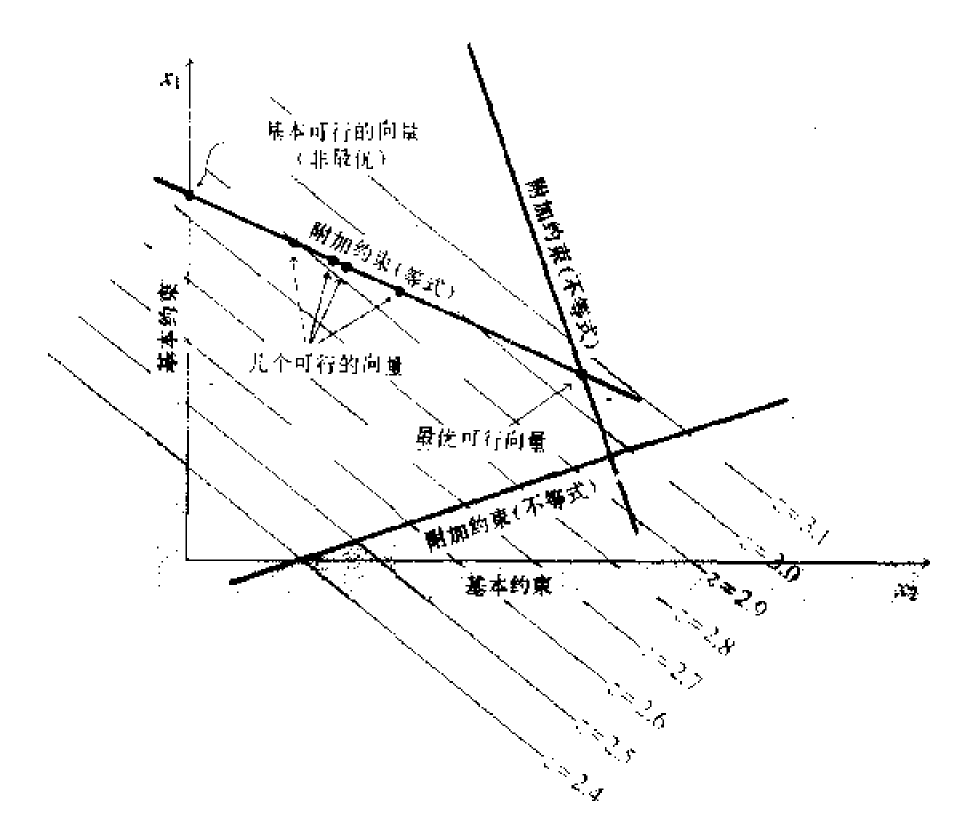
下面我们给出一个线性规划问题的具体例子。在这个例子中 $N=4, m_1=2, m_2=m_3=1$(从而有 $M=4$)：

$$\text{极大化}\quad z=x_1+x_2+3x_3-\frac{1}{2}x_4 \tag{10.8.6}$$

其中所有变量 $x$ 非负，且

$$\begin{aligned}
x_1+2x_3&\leq 740\\
2x_2-7x_4&\leq 0\\
x_2-x_3+2x_4&\geq\frac{1}{2}\\
x_1+x_2+x_3+x_4&=9
\end{aligned}\tag{10.8.7}$$

这个问题的解为(取两位有效数字) $x_1=0$, $x_2=3.33$, $x_3=4.73$, $x_4=0.95$。在本节余下的篇幅中，我们将详细介绍这个解是如何求得的。但在开始之前，我们先用图10.8.1总结一下前面提到的所有术语。




图中所示为两个独立变量的情况，目标函数 $z$ 由等高线给出。基本约束是要求 $x_1$ 和 $x_2$ 为正数，而附加约束则将问题的解划定在某一区域内（指不等式约束）或局限在降低了维数的面上（指等式约束）。可行向量要求满足所有的约束条件，基本可行向量也须位于允许区域的边界上。本节将介绍的单纯形法是在所有基本可行向量中间进行转移，直到找到最优可行向量为止。

图10.8.1　线性规划的基本概念总汇

## 10.8.1　线性规划基本定理

假设我们从待选向量的全 $N$ 维空间出发，然后（至少在想象中）把由约束条件依次所淘汰的区域"挖掉"。由于约束条件是线性的，因而由这一过程产生出来的边界将是一个平面或者一个超平面。形如式(10.8.5)的等式约束条件迫使可行区域落在降低了维数的超平面上，而不等式约束条件仅仅把可行区域划分成一些满足条件和不满足条件的小块。

当我们把所有约束条件都施加在 $N$ 维空间之后，只有两种可能发生，一种是剩下了某个可行域，另一种是根本没有可行向量存在。因为在第一种情况下，可行域是由许多超平面围成的，它在几何上就形成了一种凸多面体或单纯形（参考第10.4节）。如果确实有某个可行域存在的话，那么最优可行向量会不会完全位于其内部某处而脱离开边界呢？不会的，因为目标函数是线性的，也就是说它总会有一非零的梯度存在，而这反过来又说明，沿梯度向上总能使目标函数的值增加，直到碰到某一边界为止。

由于几何区域边界的维数比其内部的维数少一维。因此，我们可以沿投影在边界的梯度一直向上，直至到达界面的边缘；然后再沿这个边缘方向上，依此类推，不管穿过多少维数的数目，但我们最后都能到达某一点，这一点是原单纯形的某个顶点。因为它所有 $N$ 个坐标都

是已定义过的，因此这点必为同时满足某 $N$ 个等式的解，其中这 $N$ 个等式是从原来所有等式和不等式的约束集合式(10.8.2)～(10.8.5)中抽取出来的。

使原始约束中的某 $N$ 个约束能以等号满足的可行向量称为基本可行向量。若 $N>M$，则基本可行向量中至少应有 $N-M$ 个分量为零，因为我们至少还需要这么多个约束条件才能补足式(10.8.2)中的变元总数 $N$，也就是说，在基本可行解中最多有 $M$ 个分量是非零的。读者不妨用式(10.8.6)～(10.8.7)给出的例子做一下验证，它的解能以等号满足的约束条件，包括式(10.8.7)中的后三个约束条件和基本约束条件 $x_1\geqslant0$，总数达到了所要求的 4 个。

把前面两段的内容综合起来，我们可以得到**线性优化的基本定理**：若最优化可行向量存在，则必有一基本可行向量是最优的。(请注意这里的术语！)

上述定理的重要意义在于，它把优化问题化简为一个"组合"问题，即在式(10.8.2)～(10.8.5)所示的所有 $M+N$ 个约束条件中，决定最优化可行解倒底应满足其中的哪 $N$ 个约束条件的问题；而我们所要做的就是不断对各种不同的组合进行试探，并计算每种情况下的目标函数值，直到找到最优的为止。

但是盲目地使用组合方法又只能弄巧成拙，并增加数不清的计算量。丹蒂兹(Dantzig)于1948年(见[2])首先提出了针对这一问题的单纯形方法，这种方法的特点是：(i)只对某一系列组合进行试验，在这些组合中目标函数每一步都增加；(ii)一般经过几乎总是不大于 M 或 N 次数的迭代(不论 $M$ 和 $N$ 阶哪一个大)，就求出最优可行向量。这里还有一个数学史上的一个小趣闻：对于第二个性质，虽然自从单纯形法建立之后人们就早已从经验中得到了结论，但是其正确性的证明直到1982年才由斯蒂芬·斯迈尔(Stephen Smale)得出。(有关当时的报道，请读者参见[3])

### 10.8.2 关于约束标准形式的单纯形法

若在某个线性规划问题中没有式(10.8.3)或(10.8.4)形式的约束，而只有形如式(10.8.5)的等式约束和形如式(10.8.2)的非负约束，我们称该线性规划问题具有标准形式。

为了研究上的需要，不妨考察一类约束条件更严格的问题，它们具有如下的附加的性质：每一个形如式(10.8.5)的等式约束中，必须有至少一个变量的系数为正，且这个变量只在那个约束条件中出现。我们在每一约束方程中选择一个这样的变量，并以它作为变元求解该约束方程。这样选出来的变量称为**左端变量**或**基本变量**，其总数为 $M(=m_3)$个。而剩下的 $N-M$ 变量则称为**右端变量**或**非基本变量**。虽然只有在 $M\leqslant N$ 时才能得到这种**约束标准形式**，我们在下文中仍将以这类线性规划问题为出发点来考虑。

读者可能会认为这类约束标准形，未免有些过于特殊化了，它不可能包括我们想要解决的所有线性规划问题。但事实并非如此。下面我们将要介绍的就是如何将任意一个线性规划问题转化为约束标准形，以及如何将单纯形法应用于这种约束标准形式。

让我们来看一个约束标准形式问题的例子：

$$\text{极大化}\quad z=2x_2-4x_3 \tag{10.8.8}$$

其中 $x_1,x_2,x_3,x_4$非负且满足

$$\begin{aligned} x_1 &= 2-6x_2+x_3 \\ x_4 &= 8+3x_2-4x_3 \end{aligned} \tag{10.8.9}$$

此例中$N=4$，$M=2$，左端变量为$x_1$，$x_4$；右端变量为$x_2$，$x_3$。这里我们将目标函数式(10.8.8)写成了仅包含右端变量的形式。但应注意的是，这实际上并不是对具约束标准形式的目标函数的一种约束要求，因为出现在目标函数中的任何一个左端变量，都可以用式(10.8.9)或类似的式子以代数方式被右端变量替换掉。

对于任何一个具有约束标准形式的线性规划问题，我们都能立刻求出它的一个基本可行解(虽然这个基本解不必非是最优的)，方法很简单，只要将所有右端变量都置为零，满足约束的所有左端变量的值从方程(10.8.9)中即可求得。单纯形法的基本思想就是进行一系列的基本可行解的变换。在每次变换过程中，将右端变量与左端变量互调位置，而每一步中都保持当前的线性规划问题具有约束标准形，并且与原来的问题完全等价。

为了在符号表示上的方便，我们将方程(10.8.8)和(10.8.9)所包含的信息记录在如下表格中：

| | | $x_2$ | $x_3$ |
|---|---|---|---|
| $z$ | 0 | 2 | $-4$ |
| $x_1$ | 2 | $-6$ | 1 |
| $x_4$ | 8 | 3 | 4 |

(10.8.10)

读者应仔细研究式(10.8.10)，要非常清楚表中各项对应的是哪个值，还要知道对约束标准形式问题，如何在其表格形式与其表达式形式之间来回地进行转换。

单纯形法的第一步是查看表的头一行(我们称之为"z-行")，以及标有右端变量的各栏(称为"右-列")中各项的值。依次让每一右端变量从其当前的零值开始增加，同时保持其余右端变量仍为零不变，然后考察这样做的结果：目标函数值是增加了呢还是减小了？答案由z-行中各项元的符号给出。既然我们的目的是要增加函数值，因此，我们只对那些z-行项中为正值的右-列感兴趣。在式(10.8.10)中只有一列满足上述条件，它的z-行项为 2。

下一步要做的是，考察位于由第一步选出的每个z-行项下面的那些列表值。我们的问题是，在某一个左端变量变为负值(当然这是不允许的)之前，右端变量可以增加到多少呢？如果右端变量所在的列与左端变量所在行之交叉处的表中元素为正数，那么该元素将不受任何限制：相应的左端变量只会越变越大。如果任何右端变量所在列的所有表项都是正值，则目标函数无界，说明问题已经得到了答案。

若在某个值为正的z-行项下面，有一个或多个表中元素为负数，那么我们就须搞清楚，到底是哪一项首先限制了那一列的右端变量值的增加。显然，这一受限的增加量可以用右端变量所在列的元素(称为主元)来除主元素所在行的"常数列"(最左边的列)中元素而得到。若得到的数值绝对值越小说明受到限制越多，于是，由于选择了这一主元而使目标函数增加的量，应由这个数值乘以那一行对应的z-行表项值得到。对所有可能的右列重复这一过程，并求出具有最大增加量的主元，这样我们就完成了"选主元"的工作"

在上面的例子中，唯一的一个值为正的z-行表项是 2，它下面只有一个负值项，即$-6$，因此，$-6$ 即为主元，其常数列元素为 2。由于该主元允许$x_2$增加的量为$2\div|6|$，继而目标函数可因此增加$(2\times2)\div|6|$的绝对数量。

单纯形法的第三步是，给选定的右端变量一个增值，使之成为一个左端变量；同时修改

各左端变量，让主元所在行的元素减为零而使之成为一个右端变量。对上面的例子，让我们先用手算做一下：首先解主元所在行的方程，以旧的左端变量 $x_1$ 表示新左端变量 $x_2$，即：

$$x_1 = 2 - 6x_2 + x_3 \quad \rightarrow \quad x_2 = \frac{1}{3} - \frac{1}{6}x_1 + \frac{1}{6}x_3 \tag{10.8.11}$$

然后将上式代入原来的 $z$－行中，得到：

$$z = 2x_2 - 4x_3 = 2\left[\frac{1}{3} - \frac{1}{6}x_1 + \frac{1}{6}x_3\right] - 4x_3 = \frac{2}{3} - \frac{1}{3}x_1 - \frac{11}{3}x_3 \tag{10.8.12}$$

再将其代入其他所有左端变量所在的行中，这里只有 $x_4$，即：

$$x_4 = 8 + 3\left[\frac{1}{3} - \frac{1}{6}x_1 + \frac{1}{6}x_3\right] - 4x_3 = 9 - \frac{1}{2}x_1 - \frac{7}{2}x_3 \tag{10.8.13}$$

至此，由式(10.8.11)～(10.8.13)可以形成如下所示的新表格：

| | | $x_1$ | $x_3$ |
|---|---|---|---|
| $z$ | $\frac{2}{3}$ | $-\frac{1}{3}$ | $-\frac{11}{3}$ |
| $x_2$ | $\frac{1}{3}$ | $-\frac{1}{6}$ | $\frac{1}{6}$ |
| $x_4$ | 9 | $-\frac{1}{2}$ | $-\frac{7}{2}$ |

(10.8.14)

第四步是转回并重复第一步，寻找目标函数的下一可能增值。不断重复上述过程，直到 $z$－行的所有右端变量项都变成负值为止。这表明目标函数不可能再增加了。在上面给出的例子中，式(10.8.14)已经满足了这个条件，因此求解过程至此可以结束。

整个问题的解可以从最后一张表的常数列中读出。在表(10.8.14)中可以看到，目标函数的最大值为 2/3，解向量为 $x_2=1/3$，$x_4=9$，$x_1=x_3=0$。

现在让我们仔细回味一下从表(10.8.10)导出表(10.8.14)的过程。可以看出，整个过程可以完全用表格的形式总结成一系列规定的基本矩阵运算：

- 选出主元并存储之。
- 存储整个主元所在的列。
- 将表中各行(主元所在的行除外)用其自身与主元行的线性组合来替代，使其主元所在列的那一项变为零。
- 用主元的负值去除主元所在的行。
- 以存储单元内的主元值的倒数替代主元素。
- 用存储单元内主元值去除主元所在列各剩余项的存储值，并用所得结果替代主元所在列中的各剩余项。

上述一系列的操作实际上可由一个线性规划程序来完成，这个程序将在下文中给出。

至此，我们已能解决任何一个以约束标准形式给出的线性规划问题了。只有一种特殊情况在求解时可能会遇到一些麻烦，即在某一步中，常数列中的某个项元变成了零，以致于某个左端变量也为零，进而所有右端变量都成了零，这种情况下得到的解我们称之为**退化可行解**。为了使求解过程继续下去，可能需要将这个退化的左端变量与某一个右端变量进行交换，有时候这样变换可能要做好几次。

### 10.8.3 将一般问题转化为约束标准形式

关于从一般线性规划问题到约束标准形式的转化，有几种很巧妙的办法可以使转化过程变得轻而易举。

首先，需要把式(10.8.3)或式(10.8.4)形式的不等式约束条件例如式(10.8.7)中的前三个约束条件去掉。具体做法是，引入一些所谓的**松弛变量**，利用它们的非负性，将不等式转化为等式。我们将松弛变量记为 $y_i$，它们的总数有 $m_1+m_2$ 个。应注意的是，一旦将这些松弛变量引入，就要把它们与原来变量 $x_i$ 同样对待；但在求解过程结束时，将它们忽略即可

例如，在引入松弛变量之后，目标函数式(10.8.6)并未发生变化，而式(10.8.7)却变成了如下形式：

$$
\begin{aligned}
x_1 + 2x_3 + y_1 &= 740 \\
2x_2 - 7x_4 - y_2 &= 0 \\
x_2 - x_3 + 2x_4 - y_3 &= \frac{1}{2} \\
x_1 + x_2 + x_3 + x_4 &= 9
\end{aligned}
\qquad (10.8.15)
$$

(请注意松弛变量前的系数的符号由相应的原不等式的方向决定。)

第二，需确保存在一个由 $M$ 个左端变量组成的集合，这样我们就可以用约束标准形式建立一个初始表格。(换言之，需要找一个“可行的基本初始向量”。)我们采用的方法还是引入一些新的变量，这些变量称为**人工变量**，以至 $z_i$ 标记，其总为 $M$ 个。例如，在式(10.8.15)所示的每一约束等式中都引入一个人工变量，使其变为如下形式：

$$
\begin{aligned}
z_1 &= 740 - x_1 - 2x_3 - y_1 \\
z_2 &= -2x_2 + 7x_4 + y_2 \\
z_3 &= \frac{1}{2} - x_2 + x_3 - 2x_4 + y_3 \\
z_4 &= 9 - x_1 - x_2 - x_3 - x_4
\end{aligned}
\qquad (10.8.16)
$$

至此，原问题已经化为约束标准形。

读者可能会提出这样一个问题：除非所有 $z_i$ 都是零，否则式(10.8.16)与式(10.8.15)根本不等价！事实的确如此，但这里颇有一些玄妙之处。我们在求解问题时必须分两个阶段进行，第一阶段：用一个所谓的**辅助目标函数**替代原来的目标函数式(10.8.16)，即：

$$
z' \equiv -z_1 - z_2 - z_3 - z_4 = -\left(749\frac{1}{2} - 2x_1 - 4x_2 - 2x_3 + 4x_4 - y_1 - y_2 + y_3\right) \qquad (10.8.17)
$$

(其中，后一个等式是将式(10.8.16)代入得到的。)现在，我们再对式(10.8.17)所示的辅助目标函数使用单纯形法。显然，如果所有 $z_i$ 都为零，则辅助目标函数对于非负的 $z_i$ 将取得最大值。因此，我们希望单纯形法能够在第一阶段构造出一个左端变量的集合，这些左端变量仅仅从 $x_i$ 和 $y_i$ 中选取，而所有 $z_i$ 都取为右端变量。这样，我们就可以划掉所有的 $z_i$，剩下的就是只含 $x_i$ 和 $y_i$ 的约束标准形式的问题了。因此，换言之，第一阶段的任务就是构造一个初始基本可行向量。在第二阶段，我们的目标是求解由第一阶段导出的问题，但在这时要利用原来的目标函数，而不能再用辅助目标函数了。

如果在第一阶段不能对所有 $z_i$ 都取零值，又该如何处理呢？实际上，这种情况恰恰说明，该线性规划问题根本没有初始基本可行向量存在，也就是说，约束条件本身是自相矛盾的，我们只要指出这一点，就算给出了问题的解答。

下面我们将介绍，如何将第一和第二阶段所需的信息转化成表格形式。我们仍以目标函数及约束条件由式(10.8.6)～(10.8.7)给出的问题为例。

| | | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $y_1$ | $y_2$ | $y_3$ |
|---|---|---|---|---|---|---|---|---|
| $z$ | 0 | 1 | 1 | 3 | $-\frac{1}{2}$ | 0 | 0 | 0 |
| $z_1$ | 740 | $-1$ | 0 | $-2$ | 0 | $-1$ | 0 | 0 |
| $z_2$ | 0 | 0 | $-2$ | 0 | 7 | 0 | $-1$ | 0 |
| $z_3$ | $\frac{1}{2}$ | 0 | $-1$ | 1 | $-2$ | 0 | 0 | 1 |
| $z_4$ | 9 | 1 | 1 | 1 | $-1$ | 0 | 0 | 0 |
| $z'$ | $-749\frac{1}{2}$ | 2 | 4 | 2 | $-4$ | 1 | 1 | $-1$ |

(10.8.18)

这个表初看起来似乎很唬人，其实并非如此。在双线框内的表项只不过是将原问题的系数组织成表格形式而已。实际上，这些项加上 $N$、$M$，$m_1$，$m_2$，$m_3$的值就是下面的单纯形法程序所需的全部输入参数。松弛变量 $y_i$ 下面的各列简单地记录了 $M$ 约束条件中的每一个是否具有≤，≥或＝的形式；事实上只要我们在造表时是按正确的顺序填写各行内容的，这些信息及 $m_1$，$m_2$，$m_3$的值都不过是冗余信息而已。由于辅助目标的系数(最末一行)恰好是其对应列各行之和的负数，因此它们的计算非常容易。

一个单纯形法运行程序的输出需包括以下几项：(i)一个标志项，指出运行结果的类型，是有解、无解、还是有无界解；(ii)一张更新后的表格。由(10.8.18)得到输出表格为(两位有效数字)：

| | | $x_1$ | $y_2$ | $y_3$ | … |
|---|---|---|---|---|---|
| $z$ | 17.03 | $-.95$ | $-.05$ | $-1.05$ | … |
| $x_2$ | 3.33 | $-.35$ | $-.15$ | .35 | … |
| $x_3$ | 4.73 | $-.55$ | .05 | $-.45$ | … |
| $x_4$ | .95 | .10 | .10 | .10 | … |
| $y_1$ | 730.55 | .10 | .10 | .90 | … |

(10.8.19)

只需稍稍数一下 $x_i$ 和 $y_i$ 的个数就会发现，在输入表和输出表中都有 $M+1$行的项元(包括 $z$一行)，但在输出表中只有 $N+1-m_3$列(包括常数列)包含有用信息，其他列都属于可以忽略的人工变量。在输出表中，数字列的第一列包括解向量以及目标函数最大值的信息。如果某个松弛变量($y_i$)出现在表的左边，则相应的数值就表示其对应的不等式在得到充分满足时所需的量。在输出表中，非左端的变量具有零值，而具有零值的松弛变量则对应于

以等式形式被满足的约束条件。

### 10.8.4 单纯形法的程序实现

下面程序的算法是以库恩茨(Kuenzi)、特慈查契(Tzschach)和曾德(Zehnder)[4]的程序为基础而实现的。程序中除要输入 $M$、$N$、$m_1$、$m_2$、$m_3$的值外，主要的输入项是，一个二维矩阵 **a**，**a** 的存储内容是在表(10.8.18)中以双线框出的部分。这些输入值只占用 a[1..m+1][1..n+1]中的 $M+1$个行和 $N+1$个列，但在程序内部却要用到矩阵 **a** 的第 $M+2$行(用于存储辅助目标函数，这和(10.8.18)的表示完全一样)，因此程序需把 **a** 说明为 float **a，且指针必须指向允许范围为

$$a[i][k], \quad i=1..m+2, k=1..n+1 \tag{10.8.20}$$

的分配空间。如果不搞清楚这一点，后患将是无穷的。此外，还有一点不能忽略的是 a 的各行必须完全按照式(10.8.1)、(10.8.3)、(10.8.4)、(10.8.5)的顺序输入，即先输入目标函数，然后是"≤"约束条件和"≥"约束条件，最后输入"=条件约束"。

在输出项中，矩阵 **a** 的下标由两个返回的整数阵列确定。iposv[$j$]($j=1,\ldots M$)包含序数 $i$，$i$ 对应的原始变量 $x_i$ 的当前状态由 **a** 的第 $j+1$行表示，它们即为解向量中的右端变量。(**a** 的第一行当然存储的是 z-行。)若 $i$ 大于 $N$ 则表示这个 $i$ 对应的变量是松弛变量 $y_j$ 而不是 $x_i$，也即 $x_{N+j}=y_j$。同样，对 $j=1..N$，izrov[$j$] 给出了序数 $i$，它所对应的原始变量 $x_i$ 在当前状态是一个右端变量，存储存在 **a** 的第 $j+1$ 列中。$i>N$ 的意思与上面相同，所不同的是当 $i>N+m_1+m_2$时，它对应的变量表示某个人工变量或松弛变量，这个变量原先仅仅作为内部使用，而现在应予以完全忽略。

当程序运行结果为找到了一个有限解时，标志项 icase 被置为 0；当目标函数无界时，它被置为 1；如果没有解满足给定的约束条件，则 icase 被置为 −1。

下面的程序对退化可行解的情况也作了处理，因此读者不必担心这种情况会出现。此外，该程序不需存储表(10.8.18)中双线右边的各列内容，这可能也是会令使用者感到满意。不仅如此，我们还在程序中，对松弛变量的跟踪记录采取了一种更加有效的方式。

还有一点值得注意的是，程序中收敛性的检验并不如想象的那么复杂。尽管程序中对带零不等式的检验是通过对某个很小的参数 EPS 的检验来实现的，但它并没有对这个参数进行调整，以适应不同量级输入数据的变化。尽管如此，我们的程序，已经足以适合问题的需要了，因为许多问题的输入数据与单位"1"在大小的数量级上并没有相差很多。但是，如果遇到死循环的情况，则必须修改子程序 **simplx** 和 **simp2** 中的 EPS。此外，用置换变量的办法也是可以达到解决问题的目的。最后需要说明的是，与下述介绍有关的内容，读者可以参考[5]。

```
#include "nrutil.h"
#define EPS 1.0e-6              其中 EPS 是绝对精度，它应该根据用户变量的尺度做调整。
#define FREEALL free_ivector(l3,1,m);free_ivector(l2,1,m);\
    free_ivector (l1,1,n+1);

void simplx(float **a, int m, int n, int m1, int m2, int m3, int *icase, int izrov[], int iposv[])
    线性规划中的单纯形法。输入参数为 a,m,n,m1,m2和 m3,输出参数 a,icase,izrov 和 iposv,它们的定义见正文。
{
  void simp1(float **a, int mm, int ll[], int nll, int iabf, int *kp,
      float *bmax);
```

```
void simp2(float **a, int n, int l2[], int nl2, int *ip, int kp, float *q1).
void simp3(float **a, int i1, int k1, int ip, int kp);
int i,ip,ir,is,k,kh,kp,m12,nl1,nl2;
int *l1,*l2,*l3;
float q1,bmax;

if (m != (m1+m2+m3)) nrerror("Bad input constraint counts in simplx");
l1=ivector(1,n+1);
l2=ivector(1,m);
l3=ivector(1,m);
nl1=n;                                          首先将所有变量均置为右端变量
for (k=1;k<=n;k++) l1[k]=izrov[k]=k;            对索引表初始赋值
nl2=m;                                          置所有人工变量为左端变量. 并对这
for (i=1;i<=m;i++) {                            些表初始赋值
    if (a[i+1][1] < 0.0) nrerror("Bad input tableau in simplx");
    常数b_i须为非负数
    l2[i]=i;
    iposv[i]=n+i;
}
for (i=1;i<=m2;i++) l3[i]=1;                  l3在后面才能用到，但先在此处初始赋值
ir=0;
这一标志的设置表明我们已进入第二阶段，即已有了一个可行的初始解。如果起点为一个可行
解，则可以进入第二阶段
if (m2+m3) {
    ir=1;
    该标志说明我们必须从第一阶段开始
    for (k=1;k<=(n+1);k++) {                    计算辅助目标函数
        q1=0.0;
        for (i=m1+1;i<=m;i++) q1 += a[i+1][k];
        a[m+2][k] = -q1;
    }
    do {
        simp1(a,m+1,l1,nl1,0,&kp,&bmax);        求辅助目标函数的最大系数
        if (bmax <= EPS && a[m+2][1] < -EPS) {
            *icase = -1;
            辅助目标函数仍为负，而且不能进一步改进，因此无可行解存在。

            FREEALL return;
        } else if (bmax <= EPS && a[m+2][1] <= EPS) {
            m12=m1+m2+1;
            辅助目标函数为0,并不能再改进。这表明我们得到了一个可行的起始解通过
            "goto one"将所有人工变量清除，然后进入第二阶段继续求解。

            if (m12 <= m) {
                for (ip=m12;ip<=m;ip++) {
                    if (iposv[ip] == (ip+n)) {
                        simp1(a,ip,l1,
                            nl1,1,&kp,&bmax);
                        if (bmax > 0.0)
                            goto one;
                    }
                }
            }
            ir=0;                               设置参数以表明我们将进入第二阶段
            --m12;
            if (m1+1 <= m12)
                for (i=m1+1;i<=m12;i++)
                    if (l3[i-m1] == 1)
                        for (k=1;k<=n+1;k++)
                            a[i+1][k] = -a[i+1][k];
            break;                              转到第二阶段
        }
        simp2(a,m,l2,nl2,&ip,kp,&q1);           选主元(第一阶段)
        if (ip == 0) {                          由于辅助目标函数的最大值无界
            *icase = -1;                          因此不存在可行解
            FREEALL return;
```

```
        }
one:    simp3(a,m+1,n,ip,kp);                    将一个左端变量和一个右端变量进行交换
        if (iposv[ip] >= (n+m1+m2+1)) {             (第一阶段)，然后更新索引表
            for (k=1;k<=nl1;k++)
                if (l1[k] == kp) break;
            --nl1;
            for (is=k;is<=nl1;is++) l1[is]=l1[is+1];
            ++a[m+2][kp+1];
            for (i=1;i<=m+2;i++) a[i][kp+1] = -a[i][kp+1];
        } else {
            if (iposv[ip] >= (n+m1+1)) {
                kh=iposv[ip]-m1-n;
                if (l3[kh]) {
                    l3[kh]=0;
                    ++a[m+2][kp+1];
                    for (i=1;i<=m+2;i++)
                        a[i][kp+1] = -a[i][kp+1];
                }
            }
        }
        is=izrov[kp];
        izrov[kp]=iposv[ip];
        iposv[ip]=is;
    } while (ir);                               如果仍在第一阶段，返回到"do"处
}
第一阶段结束，即求一个初始可行解的工作告一段落；下面将进入第二阶段，即将该可行解进
行优化
for (;;) {
    simp1(a,0,l1,nl1,0,&kp,&bmax);              测试Z行
    if (bmax <= 0.0) {                          最优解找到，程序带值返回
        *icase=0;
        FREEALL return;
    }
    simp2(a,n,l2,nl2,&ip,kp,&q1);               选主元(第二阶段)
    if (ip == 0) {                              目标函数无界，将该信息记录下来并
        *icase=1;                                   返回
        FREEALL return;
    }
    simp3(a,m,n,ip,kp);                         对一个左端变量与一个右端变量进行
    is=izrov[kp];                                   交换(第二段阶)
    izrov[kp]=iposv[ip];
    iposv[ip]=is;
}                                               并返回作下一轮迭代
}
```

在前面的程序中，利用了下面几个实用函数。

```
#include <math.h>

void simp1(float **a, int mm, int ll[], int nll, int iabf, int *kp, float *bmax)
    确定一些元素的最大值，这些元素的索引存储在已知的列表ll中；标志iabf指示所求出的最大值是否为所有元素的
    绝对值的最大值。
{
    int k;
    float test;

    *kp=ll[1];
    *bmax=a[mm+1][*kp+1]
    for (k=2;k<=nll;k++) {
        if (iabf == 0)
            test=a[mm+1][ll[k]+1]-(*bmax);
        else
            test=fabs(a[mm+1][ll[k]+1])-fabs(*bmax);
        if (test > 0.0) {
```

```
            *bmax=a[mm+1][ll[k]+1];
            *kp=ll[k];
        }
    }
}

#define EPS 1.0e-6

void simp2(float **a, int n, int l2[], int nl2, int *ip, int kp, float *q1)    选主元素,包括退化情况在内
{
    int k,ii,i;
    float qp,q0,q;

    *ip=0;
    for (i=1;i<=nl2;i++) {
        if (a[l2[i]+1][kp+1] < -EPS) {                                       任意可能的主元?
            *q1= -a[l2[i]+1][1]/a[l2[i]+1][kp+1];
            *ip=l2[i];
            for (i=i+1;i<=nl2;i++) {
                ii=l2[i];
                if (a[ii+1][kp+1] < -EPS) {
                    q= -a[ii+1][1]/a[ii+1][kp+1];
                    if (q < *q1) {
                        *ip=ii;
                        *q1=q;
                    } else if (q == *q1) {                                   退化情况
                        for (k=1;k<=n;k++) {
                            qp = -a[*ip+1][k+1]/a[*ip+1][kp+1];
                            q0 = -a[ii+1][k+1]/a[ii+1][kp+1];
                            if (q0 != qp) break;
                        }
                        if (q0 < qp) *ip=ii;
                    }
                }
            }
        }
    }
}

void simp3(float **a, int i1, int k1, int ip, int kp)    用于交换左端变量和右端变量的一些矩阵操作(参见
{                                                           正文中的说明)
    int kk, ii;
    float piv;

    piv=1.0/a[ip+1][kp+1];
    for (ii=1;ii<=i1+1;ii++)
        if (ii-1 != ip) {
            a[ii][kp+1] *= piv;
            for (kk=1;kk<=k1+1;kk++)
                if (kk-1 != kp)
                    a[ii][kk] -= a[ip+1][kk]*a[ii][kp+1];
        }
    for (kk=1;kk<=k1+1;kk++)
        if (kk-1 != kp) a[ip+1][kk] *= -piv;
    a[ip+1][kp+1]=piv;
}
```

### 10.8.5 其他线性规划方法简述

每个包含 $N$ 个变量和 $M$ 个约束条件、且具有标准形式的线性规划问题，都有一个与其对应的**对偶问题**，该对偶问题含有 $M$ 个变量和 $N$ 个约束条件，它的表格表示本质上就是其原问题(有时称为**原始**规划)的表格表示的转置。从对偶问题的解导出原始规划的解是完全可以做到的，而且这种求解方法在计算上有时也很有用处，但它在一般情况下并不常用。

**修改的单纯形法**中有一个步骤与单纯形法完全等价，即选择哪几个左端变量和右端变量用作交换。尽管前者的计算量为与后者相比并没有很大的改进，但它在对存贮的组织方面的确与后者有不少不同之处。在计算的中间步骤，修改的单纯形法仅需一个大小为 $M\times M$ 矩阵，而不是 $M\times N$ 大小的矩阵，如果规划问题的限制条件很多，特别是对存贮量的限制也包括在其中，那么我们就可以考虑用这种修正的单纯形法。

**原始—对偶算法**以及**复合单纯法**是两种不同的方法，它们都是为了避免一般单纯形法的两个阶段而设计的；两者在同时寻找可行解和最优解方面有一定的改进。但目前来看似乎还没有什么明显的迹象表明这两种方法能很大地上优于通常情况下所使用的单纯形法。

如果目标函数和/或一个或多个约束条件，具有独立变元的非线性表达式的形式，则称该问题为**非线性规划问题**。有关这类问题的文献很多，它们都超出了本书的范围。非线性规划问题有一种特殊情况，即表达式都具有二次型式，因此称为**二次规划**。此外变量只能取整数的最优化问题称为**整数规划问题**，它是离散优化问题的一个特例。在下一节中，我们将讨论一类特殊的**离散优化**问题。

参考文献和进一步读物：

Bland, R.G. 1981, *Scientific American*, vol. 244 (June), pp.126～144. [1]

Dantzig, G.B. 1963, *Linear Programming and Extensions* (Princeton, NJ: Princeton University Press). [2]

Kolata, G. 1982, *Science*, vol. 217, p. 39. [3]

Kuenzi, H.P., Tzschach, H.G., and Zehnder, C.A. 1971, *Numerical Methods of Mathematical Optimization* (New York: Academic Press). [4]

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. 11 of *Handbook for Automatic Computation* (New York: Springer-Verlag). [5]

## 10.9 模拟退火法

**模拟退火法**(参见[1,2])作为一种适合于求解大规模的优化问题的技术，近来已引起极大的关注。特别是当优化问题有很多局部极值而全局极值又很难求出时，模拟退火法尤其有效。在实用上，它有效地"解决了"著名的**旅行推销员问题**，即在必须依次访问每一个城市(共有 $N$ 个城市)的前提下，为旅行推销员设计一条能够返回起点的最短旅程。模拟退火方法还被成功地用于设计复杂的集成电路，也就是说如何最佳地安排几十万个电路元件，使它们全部集成在一个很小的硅片上，而相互连接的线路之间的缠扰能够达到最小(参见[3,4])。尽管退火法的功效非凡，但它的算法实现却相对地简单，这一点似乎有些不可思议。

请注意，我们上面提到的两个例子都属于**组合极小化**问题。现本类问题通常也有一个目

标函数，但是函数的定义域并不是简单地由 $N$ 个连续参变量的组成的 $N$ 维空间，而是一个离散的巨大的构形空间，例如，由所有可能的城市旅行路线组成的集合，或者硅片电路元件的所有可能的分配方式的集合。构形空间中元素的数量相当巨大，根本不可能穷举，而且因为集合是离散的，我们也不可能"沿合适的方向连续下降"。因此在构形空间中，"方向"概念就没有什么意义了。

后面我们还将介绍如何在具有连续控制参数的空间中利用模拟退火法。这种应用实际上要比组合问题复杂一些，因为其中又要出现"狭长山谷"的情况。正如在下文中我们将看到的，模拟退火法的试探步骤是"随机"的；但在一个狭窄且漫长的等高线山谷中，几乎所有的随机步骤都呈向上趋势！因此，算法中需要增加一些技巧。

模拟退火的核心思想与热力学的原理颇为相似，而且尤其类似于液体流动和结晶以及金属冷却和退火的方式。在高温下，一种液体的大量分子彼此之间进行着相对自由移动。如果该流体慢慢地冷却下来，热能可动性便会消失。大量原子常常能够自行排列成行，形成一个纯净的晶体，该晶体在各个方向上都被完全有序地排列在几百万倍于单个原子大小的距离之内。对于这个系统来说，晶体状态是能量最低状态；而所有缓慢冷却的系统都可以自然达到这个最低能量状态，这可以说是一个令人惊奇的事实。实际上，如果某种液体金属被迅速冷却或被"猝熄"，那么它不会达到这一状态，而只能达到一种具有较高能量的多晶状态或非结晶状态。

因此，这一过程的本质在于缓缓地致冷，以争取充足的时间，让大量原子在丧失可动性之前进行重新分布。这就是所谓**退火**在技术上的定义，同时也是确保达到低能量状态所必需的条件。

尽管我们的比喻并不算贴切，但是迄今为止本身所讨论的所有极小化算法，确实与快速冷却猝熄有某些相关联之处。已往我们处理问题的方式都是：从初始点开始，立即沿下降方向前进，走得越远越好，似乎这样才能迅速求得问题的解。但是，正如前面常常提到的，这种方法往往只能求得局部极小点，却求不到整体小点。自然界本身的极小化算法则基于一种截然不同的方式。所谓的玻尔兹曼(Boltzmann)概率分布

$$\text{Prob}(E) \sim \exp(-E/kT) \tag{10.9.1}$$

表达了这样一种思想，即：一个处于热平衡状态且具有温度 T 的系统，其能量按照概率，分布于所有不同的能量状态 $E$ 之中。即使在很低的温度下，系统也有可能(虽然这种可能性很小)处于一个较高的能量状态。因此，相应地，系统也能够获得摆脱局部能量极小点的机会。并找到一个更好的、更接近于整体的极小点。式(10.9.1)中的参数 k(称为玻尔兹曼常数)是一个自然常数，它的作用是将温度与能量联系起来。换句话说，在有些情况下系统的能量可上升，也可下降，但是温度越低，显著上升的可能性就越小。

1953年，米特罗波利斯(Metropolis)及其合作者们首次将这种原理渗透到数值计标中，他们对一个模拟热力学系统提供了一系列选择项，并假设：系统构形从能量 $E_1$ 变化到能量 $E_2$ 的概率为 $p=\exp[-(E_2-E_1)/kT]$。很显然，如果 $E_2<E_1$，$p$ 将大于一；在这类情况下，对构形的能量变化任意指定一个概率值 $p=1$，也就是说，该系统总是取这个选择项。这种格式总是采取下降过程，偶尔采取上升步骤。目前已被公认为米特罗波利斯算法。

为了将米特罗波利斯算法应用于热力学以外的系统，必须提供以下几项基本要素：

1. 对可能的系统构形的一种描述。

2. 一个有关构形内部随机变化的生成函数，这些变化将作为“选择项”提交给该系统。

3. 一个目标函数 E(类似于能量)求解 $E$ 的极小值，即为算法所要完成的工作。

4. 一个控制参数 T(类似于温度)和一个**退火进程**，该进程用来说明系统是如何从高值向低值降低的，例如在温度 T 时每次下降步骤中要经过多少次随机的构形变化以及该步长是多大等等。应说明的是，这里“高”和“低”的含义，还有进程表的确定，都需要一定的物理知识和/或一些摸索的实验。

### 10.9.1 组合极小化：旅行推销员问题

下面是我们用“旅行推销员问题”为具体实例说明模拟退火法的应用。假设一个推销员，要去 $N$ 个分别位于$(x_i,y_i)$的城市进行推销，并于最后返回他原来所在的城市，要求每个城市只能去一次，而且所经过的路径要尽可能地短。这个问题属于一类所谓“$NP$ 一完全问题”，这类问题求出一个精确解所需的计算时间是随 $N$ 的增加以指数 exp(常数×$N$)增长的。当 $N$ 不断增大时，运行时间将迅速增加，进而导致费用高到令人难以接受的程度。旅行推销员问题也是众多极小化问题中的一种，它的目标函数 $E$ 具有多个局部极小值。在实际应用当中，常常有足够多的条件可以从多个极小值中选出一个最小的，这个最小值即使不是绝对最小，也相当接近绝对最小了。退火法的目的就是要获得这个最小值，同时又要将计算量限制在 $N$ 的低阶次的数量级上。

旅行推销员问题也是按模拟退火问题的方式进行处理的；具体如下：

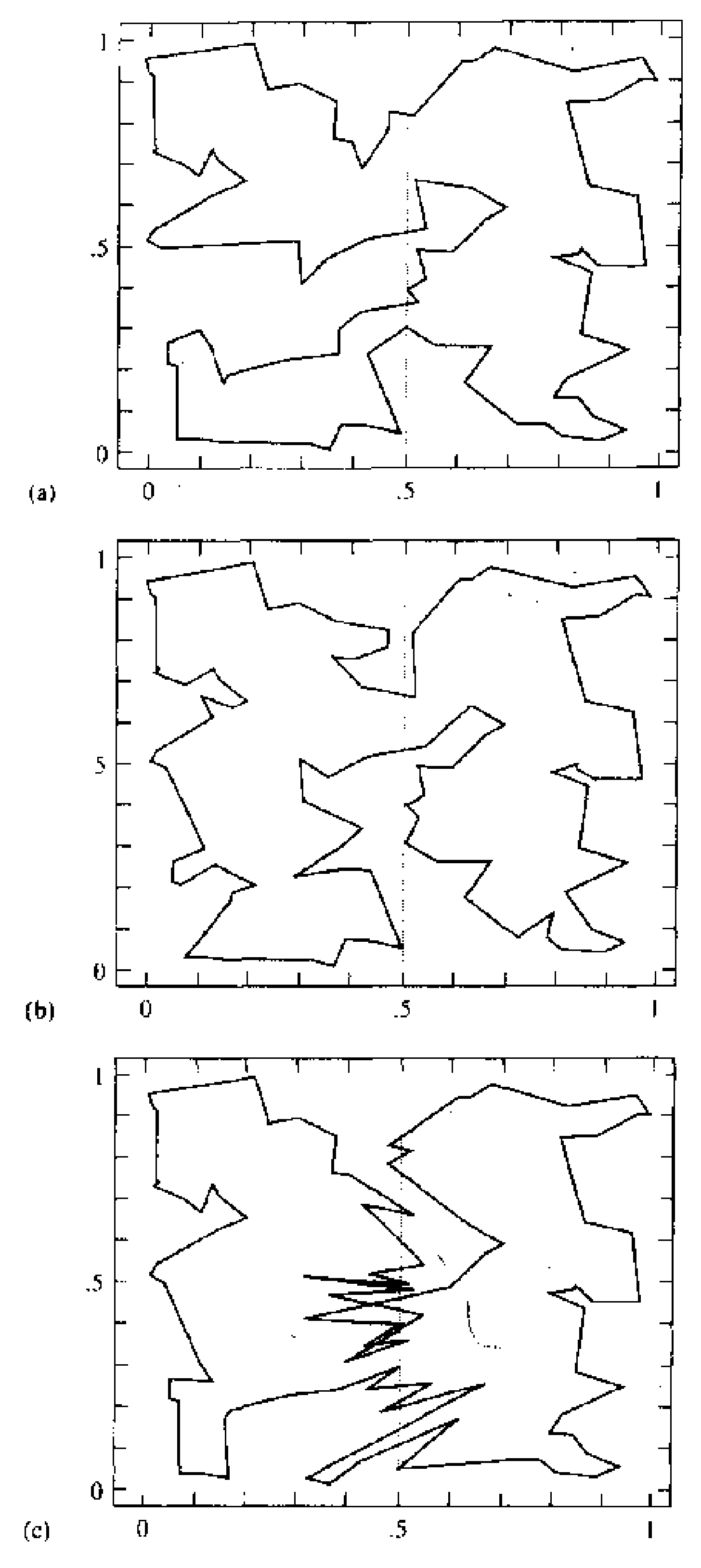
1. **构形**　将 $N$ 个城市分别标记为 $i=1...N$ 中的数，其中每个城市具有坐标$(x_i,y_i)$。一个构形就是数字1...$N$ 的一个排列，可以解释为推销员途径的城市的顺序。
2. **调整**　林(Lin)曾经提出过一种所谓“转移有效集”，这里的“转移”包括两种类型：(a)移走路径的某一段，然后对这段路径上的城市用相反次序重新进行排列，并用后者来代替前者；(b)移走某段路径，并用位于城市间的随机选取的另一段路径来取代被移走的路径。
3. **目标函数**　在旅行推销员问题的一种最简单的形式中，目标函数 $E$ 正被定义为旅途的总长度

$$E = L \equiv \sum_{i=1}^{N} \sqrt{(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2} \qquad (10.9.2)$$

这里认为第 $N+1$个点与第1个点是重合的。但是为了表明模拟退火法的灵活性，我们还要用到下面的技巧：假设推销员无端端地害怕飞越密西西比河，在这种情况下，我们对每个城市给定一个参数 $\mu_i$，如果该城市位于密西西比(Mississippi)河以东，$\mu_i$取1，若在密西西比河以西出取－1。对于目标函数 $E$，我们将其改写为：

$$E = \sum_{i=1}^{N}\left[\sqrt{(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2} + \lambda(\mu_i - \mu_{i+1})^2\right] \qquad (10.9.3)$$

由于每过一次河都将以4$\lambda$ 作为惩罚，因而现在我们设计的算法的目标，就变成了寻找尽可能回避过河的最短路径。路径长度对过河次数的相对重要性将由我们选择的 $\lambda$ 来确定。图10.9.1表明了所得的结果。显然，这种技巧可以推广到包含许多相互冲突的目的要求的极小化问题当中。




图(a)表示的是从四个随机分布的城市中间找到的一条(接近)最短路径,点线标识的是一条河流,但这是对过河没有附加惩罚项的情况。在图(b)中对过河施加的惩罚项很大,而图中所示的解本身的过河次数也相应地只有少得不能再少的两次。在图(c)中惩罚项为负,这就是说,推销员实际上成了恣意偷渡的走私者!

图10.9.1　用模拟退火法解决旅行推销员问题

4. **退火进程**　这一步需要借助试验来确定。首先要进行一些随机调整，然后利用它们来确定从转移到转移过程中将会遇到的 $\Delta E$ 值之范围。对参数 $T$ 取一初始值（这个初始值要远远大于通常所能遇到的 $\Delta E$ 的最大值），并以倍增的步长下减，每次使 $T$ 总共减少 10%。我们拿每个新的常数 $T$ 值去试各种 $100N$ 重构形，或 $10N$ 成功的重构形，无论哪个在前出现就取哪个。当实在不能再进一步减小 $E$ 时，则停止。

下面的旅行推销员程序利用了米特罗波利斯(Metropolis)算法，并展示了模拟退火技术应用于组合问题的几个主要方面。

```
#include <stdio.h>
#include <math.h>
#define TFACTR 0.9              退火进程：每步中t的下降值由该因子决定
#define ALEN(a,b,c,d) sqrt(((b)-(a))*((b)-(a))+((d)-(c))*((d)-(c)))

void anneal(float x[], float y[], int iorder[], int ncity)
    本算法用于求解在ncity个城市之间作往返旅行的最短路径，其中这ncity个城市的位置坐标存贮在数组x[1..nci
    ty]和y[1..ncity]中，数组iorder[1..ncity]表示途径城市的顺序。在输出项中，iorder中的元素将被置为数字1到
    ncity的某排列，本程序将返回它所能求出的最佳选择路径。
{
    int irbit1(unsigned long *iseed);
    int metrop(float de, float t);
    float ran3(long *idum);
    float revcst(float x[], float y[], int iorder[], int ncity, int n[]);
    void reverse(int iorder[], int ncity, int n[]);
    float trncst(float x[], float y[], int iorder[], int ncity, int n[]);
    void trnspt(int iorder[], int ncity, int n[]);
    int ans,nover,nlimit,i1,i2;
    int i,j,k,nsucc,nn,idec;
    static int n[7];
    long idum;
    unsigned long iseed;
    float path,de,t;

    nover=100*ncity;            在任何温度下，所试路径的最大次数
    nlimit=10*ncity;            在继续进行之前，成功的路径改变的最大次数
    path=0.0;
    t=0.5;
    for (i=1;i<ncity;i++) {     计算初始路径的长度
        i1=iorder[i];
        i2=iorder[i+1];
        path += ALEN(x[i1],x[i2],y[i1],y[i2]);
    }
    i1=iorder[ncity];           将路径头尾相连并结束循环
    i2=iorder[1];
    path += ALEN(x[i1],x[i2],y[i1],y[i2]);
    idum = -1;
    iseed=111;
    for (j=1;j<=100;j++) {      试验100个温度值
        nsucc=0;
        for (k=1;k<=nover;k++) {
            do {
                n[1]=1+(int) (ncity*ran3(&idum));           选择段的起始点…
                n[2]=1+(int) ((ncity-1)*ran3(&idum));       …段的结尾
                if (n[2] >= n[1]) ++n[2];
                nn=1+((n[1]-n[2]+ncity-1) % ncity);         nn为不位于当前段上的城市数
            } while (nn<3);
            idec=irbit1(&iseed);
            确定是否做段反转或段输送
            if (idec == 0) {                    做输送
```

```
            n[3]=n[2]+(int) (abs(nn-2)*ran3(&idum))+1;
            n[3]=1+((n[3]-1) % ncity);
            输送到一个不在当前路径上的某处
            de=trncst(x,y,iorder,ncity,n);       计算代价
            ans=metrop(de,t);                    做预测
            if (ans) {
                ++nsucc;
                path += de;
                trnspt(iorder,ncity,n);          输送工作结束
            }
        } else {                                 做段反转
            de=revcst(x,y,iorder,ncity,n);       计算代价
            ans=metrop(de,t);                    作预测
            if (ans) {
                ++nsucc;
                path += de;
                reverse(iorder,ncity,n);         完成段反转
            }
        }
        if (nsucc >= nlimit) break;              如果成功的转移超过次数则提前结束
    }
    printf("\n %s %10.6f %s %12.6f \n","T =",t,
        "     Path Length =",path);
    printf("Successful Moves: %6d\n",nsucc);
    t *= TFACTR;                                 退火进程
    if (nsucc == 0) return;                      如果不成功则返回
  }
}
```

```
#include <math.h>
#define ALEN(a,b,c,d) sqrt(((b)-(a))*((b)-(a))+((d)-(c))*((d)-(c)))

float revcst(float x[], float y[], int iorder[], int ncity, int n[])
```
该子程序返回的是反转某给定路径所需的代价函数值。参数中 ncity 为城市数；数组 x[1..ncity]，y[1..ncity]为这些城市的位置坐标；iorder[n..ncity]为当前路线；数组 n 的头两个元素 n[1]和 n[2]分别代表将要被反转的路径段的起点城市和终点城市。子程序的输出项 de 为反转所需的代价值，但真正的反转过程并不是由子程序执行。

```
{
    float xx[5],yy[5],de;
    int j,ii;

    n[3]=1+ ((n[1]+ncity-2) % ncity);          找出 n[1]以前的城市
    n[4]=1+ (n[2] % ncity);                    找出 n[2]以后的城市
    for (j=1;j<=4;j++) {
        ii=iorder[n[j]];                       求四个所涉及到的城市的坐标
        xx[j]=x[ii];
        yy[j]=y[ii];
    }
    de = -ALEN(xx[1],xx[3],yy[1],yy[3]);       计算断开路径段两端所需的代价
    de -= ALEN(xx[2],xx[4],yy[2],yy[4]);       以及用相反顺序重新连接所需的代价
    de += ALEN(xx[1],xx[4],yy[1],yy[4]);
    de += ALEN(xx[2],xx[3],yy[2],yy[3]);
    return de;
}
```

```
void reverse (int iorder[], int ncity, int n[])
```
该子程序的作用是执行一段路径的反转过程，输入参数 iorder[1..ncity]给出当前的路线顺序；向量 n 的前四个元素中，n[1]和 n[2]分别为将要被反转的路径的起点和终点城市，n[3]和 n[4]则分别为紧接 n[1]之前和紧随 n[2]之后的两个城市的标号，其中 n[3]和 n[4]由函数 revcst 给出。在输出端，iorder 又将被作为返回值，它的意义是 n[1]

到 n[2]路段被反转后的行程路线。

```
{
    int nn,j,k,l,itmp;

    nn=(1+((n[2]-n[1]+ncity) % ncity))/2;          为实现反转操作,必须交换这么多城市
    for (j=1;j<=nn;j++) {
        k=1 + ((n[1]+j-2) % ncity);                从段的端部开始,顺序交换城市对
        l=1 + ((n[2]-j+ncity) % ncity);            直至到达路径的中心
        itmp=iorder[k];
        iorder[k]=iorder[l];
        iorder[l]=itmp;
    }
}

#include <math.h>
#define ALEN(a,b,c,d) sqrt(((b)-(a)) * ((b)-(a))+((d)-(c)) * ((d)-(c)))

float trncst(float x[], float y[], int iorder[], int ncity, int n[])
```

该子程序返回的是输送某段给定路径所需的代价函数值。输入参数中 ncity 为城市的个数;x[1..ncity]和 y[1..ncity]为这些城市的位置坐标,数组 n 的前三个元素分别为:将要被输送的路径段的起、止点城市以及这段路径将要被插入处的标志城市(插在该城市之后)。该子程序的输出项 de 为计算出来的输送代价值,但实际的输送操作并不由该子程序执行。

```
{
    float xx[7],yy[7],de;
    int j,ii;

    n[4]=1+ (n[3] % ncity);                    找出位于n[3]之后的城市..
    n[5]=1+ ((n[1]+ncity-2) % ncity);          ..位于n[1]之前的一个城市..
    n[6]=1+ (n[2] % ncity);                    ..位于n[2]之后的一个城市..
    for (j=1;j<=6;j++) {
        ii=iorder[n[j]];                       求出六个城市的有关坐标
        xx[j]=x[ii];
        yy[j]=y[ii];
    }
    de = -ALEN(xx[2],xx[6],yy[2],yy[6]);       计算下列操作所需代价,断开n[1]到n[2]
    de -= ALEN(xx[1],xx[5],yy[1],yy[5]);       间的路径;打开n[3]和n[4]之间的空间;
    de -= ALEN(xx[3],xx[4],yy[3],yy[4]);       连接这个空间中的路径段;以及连接n[5]n[6]
    de += ALEN(xx[1],xx[3],yy[1],yy[3]);
    de += ALEN(xx[2],xx[4],yy[2],yy[4]);
    de += ALEN(xx[5],xx[6],yy[5],yy[6]);
    return de;
}

#include "nrutil.h"

void trnspt(int iorder[], int ncity, int n[])
```

该子程序的作用是执行真正的段输送操作。输入参数 iorder[1..ncity]给出当前的路径顺序;数组 n 共有6个元素,其意义分别为:n[1]和 n[2]分别代表将要被输送的路径段的起点城市和终点城市;n[3]和 n[4]为两个相邻的城市,n[1]至 n[2]路径段即将放入它们中间;n[5]和 n[6]分别为 n[1]之前和 n[2]之后的两个城市,在这六个元素中 n[4],n[5]和 n[6]由函数 trncst 给出。在输出端,iorder 将根据路径段的移动和变化作出相应的修改。

```
{
    int m1,m2,m3,nn,j,jj, * jorder;

    jorder=ivector(1,ncity);
    m1=1 + ((n[2]-n[1]+ncity) % ncity);      找出位于n[1]和n[2]间城市个数
    m2=1 + ((n[5]-n[4]+ncity) % ncity);      ...n[4]到n[5]间的城市个数
    m3=1 + ((n[3]-n[6]+ncity) % ncity);      ...n[6]和n[3]间的城市个数
    nn=1;
```

```
    for(j=1;j<=m1;j++) {
        jj=1 + ((j+n[1]-2) % ncity);                    复制所选路径段
        jorder[nn++]=iorder[jj];
    }
    if (m2>0) {
        for(j=1;j<=m2;j++) {                            复制 n[4]到 n[5]间的路径段
            jj=1+((j+n[4]-2) % ncity);
            jorder[nn++]=iorder[jj];
        }
    }
    if (m3>0) {
        for (j=1;j<=m3;j++) {                           最后,复制 n[6]到 n[3]间的路径段
            jj=1 + ((j+n[6]-2) % ncity);
            jorder[nn++]=iorder[jj];

    }
    for (j=1;j<=ncity;j++)                              将 jorder 拷贝回 iorder
        iorder[j]=jorder[j];
    free_ivector(jorder,1,ncity);
}

#include <math.h>

int metrop(float de,float t)
    该子程序为米特罗波利斯算法的程序实现,metrop 返回的是一个布尔型变量,由该变量决定是否接受一个使目标
    函数 e 产生改变量 de 的重构形。如果 de<0则 metrop=1(真);而当 de>0时,metrop 为真的概率是 exp(-de/t),这
    里才是一个由退火进程决定的温度值.
{
    float ran3(long *idum);
    static long gljdum=1;

    return de < 0.0 || ran3(&gljdum) < exp(-de/t);
}
```

### 10.9.2 模拟退火法在连续极小化问题中的应用

模拟退火法的基本思想也可以应用于具有连续 $N$ 维控制参数的极小化问题当中,例如,在某个函数 $f(\mathbf{x})$(这里 $\mathbf{x}$ 为一个 $N$ 维向量)有许多局部极小点的情况下,求解它的(理想的全局的)极小值。这时米特罗波利斯算法所需的四要素可以具体化为:1)目标函数即为 $f(\mathbf{x})$的值;2)系统状态描态即为 $N$ 维空间中的点 $\mathbf{x}$;3)类似于温度的控制参数 $T$ 以及一个使 T 逐渐降低的退火进程仍为原先的定义;4)描述构形内部随机变化的发生器即为一个从 $\mathbf{x}$ 到 $\mathbf{x}+\Delta\mathbf{x}$ 采取随机步骤的方法。

在上述四要素中问题最大的是最后一条。目前已发表的文献中[7-10],介绍了几种不同的选择 $\Delta\mathbf{x}$ 办法。但我们认为,这些方法都不算成功。问题在于"效率"二字:当局部的向下运动存在时,如果某个随机变化发生器几乎总是作出向上运动的决策,那么它的效率就很低。我们认为,一个好的发生器在等高线的"窄谷"中仍应保持高效性;当算法在接近收敛到极小点处时,它的效率也不应越变越低。在上面我们提到的几篇文献中,除[7]中介绍的方法之外,其他所有方法都表现不同程度的低效性。

下面我们将要介绍的这种方法,利用了下降单纯形法(见第10.4节)的一种修改后的形式。首先我们将米特罗波利斯算法四要素中的系统状态描述,由单个点 $\mathbf{x}$ 改为一个具有 $N+1$个点的单纯形;单纯形的"操作"和第10.4节中介绍的相同,分为反射、扩张和收缩三种。然

后我们将一个正的、呈对数分布的随机变量(与温度 $T$ 成比例)添加到存贮的函数值中.(该函数值与单纯形的每个顶点都有关联),再从每个被当作替代的新点的函数值中减去一个类似的随机变量。和普通的米特罗波利斯方法一样,这种方法总是接受一个真正的下降步骤,但有时也接受一次上升步骤。在极限过程 $T\to 0$ 中,该算法恰好变成了下降的单纯形法,并收敛到一个局部极小点。

在 $T$ 的某有限值处,单纯形将扩展到一定的规模,其大小接近于在这个温度值所能达到的区域;然后单纯形在这个区域内部做随机的滚动翻转式布朗运动,并在该过程中抽取一些新的、近似随机的点作为样本。一个区域被利用的效率与其狭窄度(对椭球状山谷,指它的主轴比率)及方向性均无关。如果温度降低得足够缓慢,那么单纯形将极有可能收缩到那个区域内,而那个区域内包含已遇到的最低相对极小。

由于在所有模拟退火法的应用场合中,"足够缓慢"一词的含意根据问题的不同可以有相当大的细微区别,因而成功与否往往取决于退火进程选择。下面几种可能性我们认为值得一试:

- 每经过 $m$ 步移动之后,将 T 减到 $(1-\epsilon)T$,这里 $\epsilon/m$ 的具体值要通过实验确定;
- 设总的移动步数为 $K$,每经过 $m$ 步移动之后将 $T$ 减到 $T=T_0(1-k/K)^\alpha$,其中 $k$ 为到目前为止所经过的步数的累加值。$\alpha$ 为一常数,可取为1,2,4等。$\alpha$ 的最佳值取决于各种深度的相对于极小的统计分布,稍大一些的 $\alpha$ 值在较低温度时,需化费的迭代次数将更多;
- 每经过 $m$ 步骤移动之后,置 $T$ 为 $\beta$ 乘 $f_1-f_b$,其中 $\beta$ 为一个阶数为1的常数,其具体值由试验确定;$f_1$ 为目前单纯形中最小的函数值;$f_b$ 曾经遇到的最佳函数。但应注意的是,$T$ 的降低幅度一次不要超过某个分数值 $\gamma$。

另一个策略方法上的问题是,当单纯形的某个顶点被放弃并让位于"永远的最佳点"时,是否需要采取重新开始的步骤,(但我们必须保证在进行这项工作时,这个"永远的最佳点"当前并不在单纯形中!)。对于有些问题重新开始(例如,只要温度降低了因子3即执行重新开始步骤)的效果极佳;而对于另外一些问题,重新开始不仅没有任何效果反而会产生负作用。上述两种截然相反的情况,我们都找到了例子可以作为例证。

将下面的程序 **amebsa** 同第10.4节中与之相应的 **amoeba** 进行比较,你会发现,参数 iter 的使用方式在两个程序中略有不同。

```
#include <math.h>
#include "nrutil.h"
#define GET_PSUM \
                    for (n=1;n<=ndim;n++) {\
                    for (sum=0.0,m=1;m<=mpts;m++)  sum += p[m][n];\
                    psum[n]=sum;}
extern long idum;                 由主程序定义和初始值
float tt;                         与 amotsa 进行通信

void amebsa(float **p, float y[], int ndim, float pb[], float *yb, float ftol,
    float (*funk)(float []),int *iter,float temptr)
```

用模拟退火与 Nelder,Mead 的下降单纯形法相结合的方法求多元函数 funk(x)的极小值,其中 x[1..ndim]为一个 ndim 维向量。作为输入参数的矩阵 p[1..ndimtl][1..ndim]共有 ndim+1行,每行均为一个 ndim 维向量,分别代表初始单纯形的各个顶点。**amebsa** 的输入项还包括矢量 y[1..ndim+1]、浮点数 ftol 以及 iter 和 temptr,其中 y 的各个元素将被初始化为函数 funk 在 p 的 ndim+1个顶点(即行)的值;ftol 为函数值所要达到的相对收敛容许限。一旦

满足要求，程序将尽早返回；iter 和 temptr 的意义分别是函数值的计算次数和温度。程序在某退火温度 temptr 处进行 iter 次函数值计算后返回。而接下来要做的事就是根据退火进程调低温度 temptr，重置 iter 并再次调用该程序（每次调用时其他参数保持不变）。如果 iter 以正值返回，则说明程序正常收敛。如果第一次调用时 yb 被初始化为一个很大的数，则 yb 和 pb[1..ndim]将依次返回已遇到的最佳函数值和最佳点（这个最佳点可以永远不是单纯形中的点。）

```
{
    float amotsa(float * *p, float y[], float psum[], int ndim, float pb[],
      float *yb, float (*funk)(float []), int ihi, float *yhi, float fac);
    float ran1(long *idum);
    int i,ihi,ilo,j,m,n,mpts=ndim+1;
    float rtol,sum,swap,yhi,ylo,ynhi,ysave,yt,ytry,*psum;

    psum=vector(1,ndim);
    tt = -temptr;
    GET_PSUM
    for (;;){
        ilo=1;                                    确定最高点(即差点)、次高点和最低点(即最佳点)
        ihi=2;
        ynhi=ylo=y[1]+tt*log(ran1(&idum));       我们所“看到”的顶点总是处于随机的热起伏状态
        yhi=y[2]+tt*log(ran1(&idum));
        if (ylo > yhi) {
            ihi=1;
            ilo=2;
            ynhi=yhi;
            yhi=ylo;
            ylo=ynhi;
        }
        for (i=3;i<=mpts;i++) {                   对单纯形中的点进行循环
            yt=y[i]+tt*log(ran1(&idum));          更多的热起伏运动
            if (yt <= ylo) {
                ilo=i;
                ylo=yt;
            }
            if (yt > yhi) {
                ynhi=yhi;
                ihi=i;
                yhi=yt;
            } else if (yt > ynhi) {
                ynhi=yt;
            }
        }
        rtol=2.0*fabs(yhi-ylo)/(fabs(yhi)+fabs(ylo));    计算从最高点到最低点的范围，若合乎要求，
                                                         则返回
        if (rtol < ftol || *iter < 0) {           若返回，将最佳点和最佳值放入槽1中
            swap=y[1];
            y[1]=y[ilo];
            y[ilo]=swap;
            for (n=1;n<=ndim;n++) {
                swap=p[1][n];
                p[1][n]=p[ilo][n];
                p[ilo][n]=swap;
            }
            break;
        }
        *iter -= 2;                               开始新的一轮迭代，首先从高点通过单纯形的表面，以因子
                                                  -1做外插，即从高点对单纯形进行反射。
        ytry=amotsa(p,y,psum,ndim,pb,yb,funk,ihi,&yhi,-1.0);
        if (ytry <= ylo) {                        得到了比最佳点还要好的结果，因此用因子2再做一次外推
            ytry=amotsa(p,y,psum,ndim,pb,yb,funk,ihi,&yhi,2.0);
```

```
        } else if (ytry >= ynhi) {                        反射后的点不如次高点，因此需要找一个中间的较低点
                                                          即做一次一维收缩
            ysave=yhi;
            ytry=amotsa(p,y,psum,ndim,pb,yb,funk,ihi,&yhi,0.5);
            if (ytry >= ysave) {                          似乎无法摆脱高点，最好围绕最低点也即最佳点进行收缩
                for (i=1;i<=mpts;i++) {
                    if (i != ilo) {
                        for (j=1;j<ndim;j++)
                            psum[j]=0.5*(p[i][j]+p[ilo][j]);
                            p[i][j]=psum[j];
                        }
                        y[i]=(*funk)(psum);
                    }
                }
                *iter -= ndim;
                GET_PSUM                                  重新计算 psum
            }
        } else ++(*iter);                                 纠正计数器
    }
    free_vector(psum,1,ndim);
}


#include <math.h>
#include "nrutil.h"

extern long idum;                                         在主程序中定义和初始赋值
extern float tt;                                          在 amobsa 中定义

float amotsa(float **p, float y[], float psum[], int ndim, float pb[],
    float *yb, float (*funk)(float []), int ihi, float *yhi, float fac)
从高点通过单纯形的表面，以因子 fac 作外推，如果得到的新点较好，则用它取代高点。
{
    float ran1(long *idum);
    int j;
    float fac1,fac2,yflu,ytry,*ptry;

    ptry=vector(1,ndim);
    fac1=(1.0-fac)/ndim;
    fac2=fac1-fac;
    for (j=1;j<=ndim;j++)
        ptry[j]=psum[j]*fac1-p[ihi][j]*fac2;
    ytry=(*funk)(ptry);
    if (ytry <= *yb) {                                    存储至今最好的
        for(j=1;j<=ndim;j++) pb[j]=ptry[j];
        *yb=ytry;
    }
    yflu=ytry-tt*log(ran1(&idum));                        我们曾经对所有的当前顶点添加热起伏运动，
    if (yflu < *yhi) {                                    但这里我们要减少热起伏，目的是给单纯形
        y[ihi]=ytry;                                      个热布朗运动；就象接受任何提议的变化一样
        *yhi=yflu;
        for (j=1;j<=ndim;j++) {
            psum[j] += ptry[j]-p[ihi][j];
            p[ihi][j]=ptry[j];
        }
    }
    free_vector(ptry,1,ndim);
    return yflu;
}
```

模拟退火法在优化方法中将处于什么样的地位、扮演什么样的角色?对于这个问题目前还没有足够的实践经验来明确回答它。但是这种方法确有几个极为惹人注目的特点,与其他优化技术相比显得尤其突出:

首先,这种方法不象“急功近利的小人”那样表现“贪婪”,也就是说,它能迅速求出某些局部极小值,但那些实际上是不利的极小值并不会轻易上当。假如给定一些足够一般的重构形,这种方法会在深度小于 $T$ 的局部极小值之间自由地游动。当 $T$ 值降低,这些具备频繁访问条件的极小值的数目也将逐渐减少。

第二点,这种方法中有关构形的判定是趋向于以一种合乎逻辑的顺序进行的。当控制参数 $T$ 很大时,那些引起最大能量差异的诸多变化将被筛去。$T$ 减小时,这些判定就变得更恒定,相应地,算法将注意力就更多地转向解的较精细的改进方面。例如,在旅行推销员问题中,如果密西比河迂回于旅途之中,而且 $\lambda$ 值很大,那么,在很高的 $T$ 值下,只能做出跨河两次的判定,而位于跨河两岸的一些特定路径只能在稍后的步骤中才能确定下来。

以热力学原理为借鉴的方法远不止我们所讨论的这些内容,它们还可以推广到更广泛的应用领域,我们可以定义一些与特定热量和熵类似的量,这些量对于监测算法向某个可接受解逐步接近的过程是很有帮助的。关于这些内容的讨论读者可参考[1]中的介绍。

参考文献和进一步读物:


Kirkpatrick,S,Gelatt,C. D,and Vecchi,M. P. 1983,*Science*,Vol. 220,pp. 671~680. [1]

Kirkatrick,S. 1984,*Joumal of Statistical Physics*,vol. 34,pp. 975~986. [2]

Vecchi,M. P. and Kirkpatrick,S. 1983,*IEEE Transactions on Computer Aided Desing*,vol. CAD-2,PP. 215~222. [3]

Otten,R. H. J. M. ,and van Ginneken,L. P. P. P. 1989,*The Annealing Aigorithm* (Boston:Kluwer) [contains many references to the literature]. [4]

Metropolis,N,Rosenblu,A,Rosenbluth,M,Teller A,and Teller,E. 1953. *Joumal of Chenical Physics*,vol. 21,pp. 1087~1092. [5]

Lin,S. 1965. *Bell System Technical Joumal*,vol. 44,pp. 2245~2269. [6]

Vanderbilt,D,and Louie,S. G. 1984,*Joumal of Computational Physics*,vol. 56,pp. 259~271. [7]

Bohachevsky,I. O,Johnson,M. E,and Stein,M. L. 1986,*Technometrics*,vol. 56,pp. 259~271. [8]

Corana,A. Marchesi,M,Martinl,C,and Ridella,S,1987,*ACM Transactions on Mathematical Software*,vol. 13,pp. 262~280. [9]

Belisle,C. J. P. Romeijn,H. E. ,and Smith,R. L. 1990,Technical Report 90~25,Department of Industrial and Operations Engineering, University of Michigan,submitted to *Mathematical Programming*. [10]

# 第十一章　特征系统

## 11.0　引言

一个 $N\times N$ 阶矩阵 $\mathbf{A}$ 如果满足下式，则它具有一个特征量 $\mathbf{x}$ 和相应的特征值 $\lambda$

$$\mathbf{A}\cdot\mathbf{x}=\lambda\mathbf{x} \tag{11.0.1}$$

显然，特征量 $\mathbf{x}$ 的任意倍数也是特征向量，但我们不认为这样的倍数是不同的特征向量（零向量也不认为是特征向量）。显而易见，仅当在满足下式时，式(11.0.1)才成立：

$$\det|\mathbf{A}-\lambda\mathbf{1}|=0 \tag{11.0.2}$$

如果展开上式，就可得到一个 $N$ 阶多项式，其根就是特征值。这也证明了总存在 $N$ 个特征值（不一定是不同的）。来自于重根的相同的特征值称为**退化**。用特征方程求根的方法来求特征值，这不是好的计算方法。本章我们将学习一些更好的方法，以及寻找对应的特征量的有效方法。

上面两式同样说明在 $N$ 个特征值中，任意一个特征值都有一个（不一定是不同的）对应的特征向量：如果 $\lambda$ 确定为某特征值，则矩阵 $\mathbf{A}-\lambda\mathbf{1}$ 是奇异的，而我们知道，每个奇异矩阵在它的零空间里至少有一个非零向量（参见第2.9节关于奇异值分解）。

如果将式(11.0.1)两边加上 $\tau\mathbf{x}$，则很容易发现，任意矩阵加上一个常数 $\tau$ 乘以单位矩阵后得到的矩阵，就将相应的特征值改变或平移了一个常数 $\tau$，在这种平移下，特征向量不变。我们将会看到，这种平移特征是很多计算特征值算法中的重要内容，我们同样看到，零特征值没有什么特殊的意义。任何特征值都可通过平移成为零特征值，或者零特征值平移成为非零特征值。

### 11.0.1　定义和基本事实

一个矩阵如果等于它自身的转置，则称为是**对称**的：

$$\mathbf{A}=\mathbf{A}^T \quad 或 \quad a_{ij}=a_{ji} \tag{11.0.3}$$

如果一个矩阵等于自身转置的复共轭（它的**埃尔米特共轭**，用“ † ”表示），则称它为**埃尔米特**(Hermitian)**的**或**自轭的**

$$\mathbf{A}=\mathbf{A}^\dagger \quad 或 \quad a_{ij}=a_{ji}{}^* \tag{11.0.4}$$

如果矩阵的转置等于它的逆，则称它为**正交**的：

$$\mathbf{A}^T\cdot\mathbf{A}=\mathbf{A}\cdot\mathbf{A}^T=\mathbf{1} \tag{11.0.5}$$

如果矩阵的埃尔特共轭等于它的逆，则称它为**酉矩阵**。最后，如果一个矩阵与它的埃尔米特共轭可以交换，则称它为**正规的**：

$$\mathbf{A}\cdot\mathbf{A}^\dagger=\mathbf{A}^\dagger\cdot\mathbf{A} \tag{11.0.6}$$

对于实矩阵来说，埃尔米特性和对称性是同一的。酉正性和正交性是同一的，而且这两类不同的矩阵都是正规的。

矩阵的“埃尔米特性”在矩阵特征值问题中的重要性，在于埃尔米特矩阵的特征值都是实的。相反地，实非对称矩阵的特征值可能包含实值，也可能包含成对的复共轭值。而对于埃尔米特的复矩阵来说，其特征值一般都是复的。

“正规的”这一概念在特征值问题中是很重要的，因为具有非退化特征值（不同的）的正规矩阵的特征向量是完备正交的，它们可张成一个 $N$ 维的向量空间。对于具有退化特征值的正规矩阵，我们可以将对应于退化特征值的特征向量进行线性组合来代替它的特征向量。利用这一原则，我们总可以从格兰姆-施密特(Gram-Schmidt)正交化方法（参阅任意一本线性代数书），找到一组完备正交的特征向量，就象非退化的情形一样。若矩阵各列是正交的特征向量集，则显然这矩阵是酉矩阵。一种特殊的情形是，由实对称矩阵的特征向量构成的矩阵是正交的，因为该矩阵的特征向量都是实的。

如果一个矩阵是非正规的，如任何随机非对称的实矩阵，则一般不能找到任何一组规一化正交的特征向量，甚至不能找到任何一对正交的特征向量（除非出于偶然）。这 $N$ 个非规一化正交的特征向量“常常”能张成一个 $N$ 维向量空间，但不是总能这样做，也就是说，这些特征向量不总是完备的。这样的矩阵称为是**亏损的**。

## 11.0.2 左特征向量和右特征向量

非正规矩阵的特征向量不能保证它们之间是正交的，但它们却与另一组我们将要定义的向量有正交的关系。迄今为止，我们提到的特征向量都是满足式(11.0.1)与矩阵 $\mathbf{A}$ 右乘的列向量。更准确地讲，应称这些向量为**右特征向量**。我们同样可以发现，满足下式的左乘以矩阵 $\mathbf{A}$ 的行向量：

$$\mathbf{x}\cdot\mathbf{A}=\lambda\mathbf{x} \tag{11.0.7}$$

这样的向量称为**左特征向量**。将式(11.0.7)两边取转置，我们可得，左特征向量就是 $\mathbf{A}$ **转置**的右特征向量的转置。现在与式(11.0.2)比较，并且利用矩阵的行列式等于其转置的行列式这一事实，显然左右的特征值是恒等的。

如果矩阵 $\mathbf{A}$ 是对称的，则左、右特征向量恰好是互为转置，也就是说，分量的数值是相同的。相似地，如果矩阵是自轭的，则左、右特征向量是互为埃尔米特共轭的。对于一般的非正规情形，我们有如下的计算：令 $\mathbf{X}_R$ 表示为由右特征向量按列排列形成的矩阵，$\mathbf{X}_L$ 表示为由左特征向量按行排列形成的矩阵，则式(11.0.1)和(11.0.7)可重写成

$$\begin{aligned}\mathbf{A}\cdot\mathbf{X}_R&=\mathbf{X}_R\cdot\operatorname{diag}(\lambda_1\ldots\lambda_N)\\ \mathbf{X}_L\cdot\mathbf{A}&=\operatorname{diag}(\lambda_1\ldots\lambda_N)\cdot\mathbf{X}_L\end{aligned} \tag{11.0.8}$$

将前一式两边左乘以 $\mathbf{X}_L$，将后一式两边右乘 $\mathbf{X}_R$，两式相减得

$$(\mathbf{X}_L\cdot\mathbf{X}_R)\cdot\operatorname{diag}(\lambda_1\ldots\lambda_N)=\operatorname{diag}(\lambda_1\ldots\lambda_N)\cdot(\mathbf{X}_L\cdot\mathbf{X}_R) \tag{11.0.9}$$

这说明左右特征向量之点积构成的矩阵与特征值的对角矩阵可以交换。但是，只有对角矩阵本身和具有相异元素的对角矩阵才可以交换。因而，如果特征值是非退化的，则左特征向量除去所对应的右特征向量以外，应是与所有右特征向量相互正交的，反之亦然。通过归一化，对于任何非退化特征值的矩阵来说，相对应的左、右特征向量点积总可以成为单位一。

如果特征值是退化的，则与退化特征值相对应的左、右特征向量都必须进行线性组合，才能得到分别与左或右特征向量的正交性。这总可以通过一种类似于格兰姆-施密特正交化方法的过程来实现。于是可以调整归一化的方法，就可以使对应的左右特征向量的非零点积

成为单位一。如果对应的左右特征向量的点积为零，则说明这样的特征向量是不完备的！注意，不完备的特征向量只有在退化特征值的情况下出现，但在这种情况下，也并不总是如此（事实上，对于"正规的"矩阵就决不会发生）。详细讨论参见文献[1]。

不管是退化或非退化的情形，最后对于非零点积的归一化产生如下结果：如果逆矩阵存在的话，左特征向量按行构成的矩阵是右特征向量按列构成的矩阵的逆。

### 11.0.3 矩阵的对角化

将式(11.0.8)的第一式左乘 $\mathbf{X}_L$，并利用 $\mathbf{X}_L$ 和 $\mathbf{X}_R$ 互为逆矩阵，得到

$$\mathbf{X}_R^{-1} \cdot \mathbf{A} \cdot \mathbf{X}_R = \mathrm{diag}(\lambda_1, \ldots \lambda_N) \tag{11.0.10}$$

这是对矩阵 **A** 的**相似变换**的一种特殊情形：

$$\mathbf{A} \quad \rightarrow \quad \mathbf{Z}^{-1} \cdot \mathbf{A} \cdot \mathbf{Z} \tag{11.0.11}$$

其中 **Z** 是某个变换矩阵。相似变换在特征计算中起着决定性的作用，因为相似变换不改变一个矩阵的特征值。这很容易从以下推导看出

$$\begin{aligned} \det|\mathbf{Z}^{-1} \cdot \mathbf{A} \cdot \mathbf{Z} - \lambda \mathbf{1}| &= \det|\mathbf{Z}^{-1} \cdot (\mathbf{A} - \lambda \mathbf{1}) \cdot \mathbf{Z}| \\ &= \det|\mathbf{Z}| \det|\mathbf{A} - \lambda \mathbf{1}| \det|\mathbf{Z}^{-1}| \\ &= \det|\mathbf{A} - \lambda \mathbf{1}| \end{aligned} \tag{11.0.12}$$

式(11.0.10)说明，任何具有完备特征向量的矩阵(包括所有正规矩阵和"大多数"随机非正规矩阵)，可以通过相似变换而对角化，实现对角化的变换矩阵之列就是右特征向量，而它的逆矩阵之行就是左特征向量。

对实对称矩阵来说，特征向量是实的并且是规一正交的，因而变换矩阵是正交的。所以这一相似变换也是一种下列形式的**正交变换**：

$$\mathbf{A} \quad \rightarrow \quad \mathbf{Z}^T \cdot \mathbf{A} \cdot \mathbf{Z} \tag{11.0.13}$$

实的非对称性矩阵可以通过它们的完备特征向量进行对角化，但变换矩阵不一定是实的。实际情况是，一个实的相似变换可以"基本上"实现对角化。它可以将矩阵简化到沿主对角线只有一些两行两列的小块，而其它部分元素皆为零。每一个两行两列的小块对应一对复共轭的复特征值。在这一章的以后部分将会看到利用这一思想的一些程序。

几乎所有现代特征系统的计算程序的"重大策略"，都是通过一系列相似变换，将矩阵 **A** 接近对角形式：

$$\begin{aligned} \mathbf{A} \quad \rightarrow \quad \mathbf{P}_1^{-1} \cdot \mathbf{A} \cdot \mathbf{P}_1 \quad &\rightarrow \quad \mathbf{P}_2^{-1} \cdot \mathbf{P}_1^{-1} \cdot \mathbf{A} \cdot \mathbf{P}_1 \cdot \mathbf{P}_2 \\ \rightarrow \quad \mathbf{P}_3^{-1} \cdot \mathbf{P}_2^{-1} \cdot \mathbf{P}_1^{-1} \cdot \mathbf{A} \cdot \mathbf{P}_1 \cdot \mathbf{P}_2 \cdot \mathbf{P}_3 \quad &\rightarrow \quad \text{等} \end{aligned} \tag{11.0.14}$$

如果最终得到了对角形式，则特征向量是总变换的列向量

$$\mathbf{X}_R = \mathbf{P}_1 \cdot \mathbf{P}_2 \cdot \mathbf{P}_3 \cdot \ldots \tag{11.0.15}$$

有时，我们不需要一直变换成对角形式。例如，我们只关心特征值而不是特征向量，那么，将矩阵 **A** 变成三角形式已经足够了，即对角线以下(或以上)的元素都为零。在这种情况下，对角元素已经是特征值了，这可以通过将式(11.0.2)，用子行列式展开来验证。

有两组不同的技术可以用来实施上述重大策略式(11.0.14)。实际上，它们联合起来使用效果最好，所以大多数现代特征系统程序都使用了这两组技巧。第一组技巧是，由一系列用来完成不同特殊功能的"原子"变换 $\mathbf{P}_i$ 构成，例如将某一特定非对角元素零化(雅可比(Jacobi)变换，第11.1节)；或将某一特定的整行或整列零化(豪斯贺德(Householder)变换，

第11.2节;消去法,第11.5节)。一般说来,有限次这样的简单变换不能将整个矩阵对角化。于是有两种选择:或者用有限次这种变换完成大部分约化工作(例如,转化成一些殊如**三对角形**或**海森伯格**(Hessenberg)的特殊形式),然后再使用将要提到的第二组技术;或者反复应用这一系列有限次简单的变换,直到矩阵与对角形式之间的差别可忽略。这后一种方法在概念上最简单,所以我们将在下一节中讨论它。尽管如此,对于 $N$ 大于~10,这种方法在计算上的非有效性大约有常数因子~5,所以效率很低。

第二种更加精巧的技术称为**因子分解法**。假设矩阵 $\mathbf{A}$ 能被分解成一个左因子 $\mathbf{F}_L$ 和一个右因子 $\mathbf{F}_R$,则

$$\mathbf{A} = \mathbf{F}_L \cdot \mathbf{F}_R \cdot \quad \text{或等价地} \quad \mathbf{F}_L^{-1} \cdot \mathbf{A} = \mathbf{F}_R \tag{11.0.16}$$

如果我们将这两因子交换相乘,并利用式(11.0.16)中的第二式,得到

$$\mathbf{F}_R \cdot \mathbf{F}_L = \mathbf{F}_L^{-1} \cdot \mathbf{A} \cdot \mathbf{F}_L \tag{11.0.17}$$

上式可以看作是对 $\mathbf{A}$ 进行了一个相似变换,其变换矩阵是 $\mathbf{F}_L$!在第11.3节和第11.6节中我们将讨论体现这种思想的 $QR$ 方法。

分解因子法同样地不能经过有限次变换就准确地收敛。但某一些较好的因子分解法确能更快捷而可靠地收敛,并且当伴随了一个使用简单相似变换的适当的初始简化时,它们则是应选择的方法。

### 11.0.4 "成品化特征系统程序的特征系统软件包"

现在,读者可能已经意识到,特征系统的求解是个很复杂的问题。确实如此。这正是本书所涉及的少数几个主题中的一个。关于这个主题,我们建议使用已成品化的程序。本章的目的是,要正确地告诉读者这些成品化程序的内部工作情况,以便使你在使用它们时能审慎地作出选择,并且当出现错误时能明智地予以判断。

当今几乎所有的成品化程序都可以追溯到它们的前身,即 Wilkinson 和 Reinsch 发表的《自动计算手册,第二卷,线性代数》(*Handbook for Automatic Computation, Vol. II, Linear Algebra*)[2]中的程序。这一本包含了许多位作者论述的出色参考书,它们是这一领域的圣经。《手册》中公开部分的 FORTRAN 程序是 EISPACK 程序集[3]。

本章中程序都是从《手册》或 EISPACK 中的程序转译过来的,所以理解了这些程序将会有助于增进对那些典型软件包的理解。

IMSL[4]和 NAG[5]都提供了用 FORTRAN 编写的专用程序的实施,这些程序基本上是《手册》中的程序。

一个好的"特征系统软件包"将提供进行如下所需计算的不同的程序或途径:

- 所有的特征值但不求特征向量
- 所有的特征值和某些对应的特征向量
- 所有的特征值和所有对应的特征向量

区别这些的目的是为了节省计算时间和存储量,计算那些并不需要的特征向量是很浪费工时的。人们经常只需知道几个最大的或几个绝对值最大的、或几个负的特征值所对应的特征向量。如果只需要不到总数四分之一的特征向量,那么采用计算这些部分特征向量的方法要比计算所有特征向量的方法更为有效。

一个好的特征系统软件包除了可以分别解决上述不同问题外,还应分别针对下列不同

类型的矩阵：

- 实、对称、三对角型矩阵
- 实、对称、带状(只有少数次对角或超对角元素是非零的)矩阵
- 实、对称矩阵
- 实、非对称矩阵
- 复、埃尔米特矩阵
- 复、非埃尔米特矩阵

区分这样的目的同样是为了节省时间和存储量。它是通过采用在任何特定应用中都适用的最少通用程序来达到这个目的的。

下面，我们给出实施有关下述问题的优秀程序所处的章节。

- 计算实对称三对角型矩阵的所有特征值和特征向量(第11.3节)
- 计算实对称矩阵的所有特征值和特征向量(第11.1节～第11.3节)
- 计算复埃尔米特矩阵的所有特征值和特征向量(第11.4节)
- 计算实非对称矩阵的所有特征值，不计算特征向量(第11.5节～第11.6节)

我们也将在第11.7节中讨论如何用逆迭代的方法求得非对称矩阵的某些特征向量。

### 11.0.5 广义的和非线性特征值问题

许多特征系统程序包还可以处理所谓的**广义特征问题**：

$$\mathbf{A}\cdot\mathbf{x}=\lambda\mathbf{B}\cdot\mathbf{x} \tag{11.0.18}$$

其中 $\mathbf{A}$ 和 $\mathbf{B}$ 都是矩阵。在很多这类问题中，$\mathbf{B}$ 是非奇异的，它能用下面等式处理：

$$(\mathbf{B}^{-1}\cdot\mathbf{A})\cdot\mathbf{x}=\lambda\mathbf{x} \tag{11.0.19}$$

通常 $\mathbf{A}$ 和 $\mathbf{B}$ 是对称的，并且 $\mathbf{B}$ 是正定矩阵。在式(11.0.19)中矩阵 $\mathbf{B}^{-1}\cdot\mathbf{A}$ 是非对称矩阵，但可以通过采用第2.9节中乔莱斯基(Chlisky)分解 $\mathbf{B}=\mathbf{L}\cdot\mathbf{L}^T$ 方法，将它转换成对称特征值问题。式(11.0.18)乘以 $\mathbf{L}^{-1}$，得

$$\mathbf{C}\cdot(\mathbf{L}^T\cdot\mathbf{x})=\lambda(\mathbf{L}^T\cdot\mathbf{x}) \tag{11.0.20}$$

其中

$$\mathbf{C}=\mathbf{L}^{-1}\cdot\mathbf{A}\cdot(\mathbf{L}^{-1})^T \tag{11.0.21}$$

矩阵 $\mathbf{C}$ 是对称的并且与原问题式(11.0.18)有相同的特征值。它的特征函数是 $\mathbf{L}^T\cdot\mathbf{x}$。构成 $\mathbf{C}$ 矩阵的有效方法是，首先求解方程

$$\mathbf{Y}\cdot\mathbf{L}^T=\mathbf{A} \tag{11.0.22}$$

得矩阵 $\mathbf{Y}$ 的下三角元素，然后，对于对称矩阵 $\mathbf{C}$ 的下三角元素，可通过求解

$$\mathbf{L}\cdot\mathbf{C}=\mathbf{Y} \tag{11.0.23}$$

另一类广义的标准特征值问题是特征值 $\lambda$ 中的非线性问题，例如，

$$(\mathbf{A}\lambda^2+\mathbf{B}\lambda+\mathbf{C})\cdot\mathbf{x}=0 \tag{11.0.24}$$

这类问题可以通过引入一个附加的未知特征向量 $\mathbf{y}$，并求解 $2N\times 2N$ 阶特征系统，而转换成线性问题。这个特征系统是

$$\begin{pmatrix}0 & \mathbf{1}\\ -\mathbf{A}^{-1}\cdot\mathbf{C} & -\mathbf{A}^{-1}\cdot\mathbf{B}\end{pmatrix}\cdot\begin{pmatrix}\mathbf{x}\\ \mathbf{y}\end{pmatrix}=\lambda\begin{pmatrix}\mathbf{x}\\ \mathbf{y}\end{pmatrix} \tag{11.0.25}$$

这类技术归纳为 $\lambda$ 的高阶问题。$M$ 阶多项式产生线性的 $MN\times MN$ 阶特征系统(参见[7])。

参考文献和进一步读物：

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 6. [1]

Wilkinson, J. H., and Reinsch, C. 1971, *Linear Algebra*, vol. 11 of *Handbok for Automatic Computation* (New York: Springer-Verlag). [2]

Smith, B. T., et al. 1976, *Matrix Eigensystem Routines -- EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag). [3]

*IMSL Math/Library Users Manual* (IMSL Inc., 2500 CityWest Boulevard, Houston TX 77042). [4]

*NAG Fortran Library* (Numerical Algorthms Group, 256 Banbury Road, Oxford OX27DE, U.K.). Chapter F02. [5]

Golub, G. H., and Van Loan, C. F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), § 7.7. [6]

Wilkinson, J. H. 1965, *The Algebraic Eigenvalue Problem* (New York: Oxford University Press). [7]

## 11.1 对称矩阵的雅可比变换

雅可比方法是由一系列形式(11.0.14)的正交相似变换组成的。每一步变换(**雅可比旋转**)是用来将某一非对角元素零化的平面旋转。相继的变换会把以前已经零化的元素再赋值，尽管如此，非对角元素会变得越来越小，直到在机器的精度内矩阵对角化。将所有变换累乘起来，就得到特征向量的矩阵(如式(11.0.15)的形式)，而最终对角化矩阵的对角元素就是特征值。

对所有实对称矩阵来说，雅可比方法是非常简单易行的。但当矩阵维数大于10以后，这种算法将会以一个显著的常数因子慢下来(与第11.3节中将要介绍的 $QR$ 方法相比较)。不管怎样，雅可比方法比大多数更有效的方法要简单得多。对于计算代价不是考虑的主要问题的场合，并且维数适度的矩阵，我们推荐使用这种方法。

基本的雅可比旋转 $\mathbf{P}_{pq}$ 是如下形式的矩阵

$$\mathbf{P}_{pq} = \begin{bmatrix} 1 & & & & & & \\ & \cdots & & & & & \\ & & c & \cdots & s & & \\ & & \vdots & 1 & \vdots & & \\ & & -s & \cdots & c & & \\ & & & & & \cdots & \\ & & & & & & 1 \end{bmatrix} \tag{11.1.1}$$

其除了在行列数都为 $p$ 和 $q$ 的对角元素为两个元素 $c$ 外，其它对角元素为1。除去两个元素 $s$ 和 $-s$ 外，所有其它非对角元素都为零。$c$ 和 $s$ 分别是旋转角 $\varphi$ 的余弦和正弦，故 $c^2+s^2=1$。

一个式(11.1.1)的平面旋转按如下方式变换矩阵 $\mathbf{A}$

$$\mathbf{A}' = \mathbf{P}_{pq}^T \cdot \mathbf{A} \cdot \mathbf{P}_{pq} \tag{11.1.2}$$

$\mathbf{P}_{pq}^T \cdot \mathbf{A}$ 只改变 $\mathbf{A}$ 的 $p$ 行和 $q$ 行，而 $\mathbf{A} \cdot \mathbf{P}_{pq}$ 只改变 $\mathbf{A}$ 的 $p$ 列和 $q$ 列。注意，这里用的下标 $p$ 和 $q$ 并不是指 $\mathbf{P}_{pq}$ 中的元素，而是用以表明矩阵是哪一种类型的旋转，即标明它所作用那些行

和列。因此，式(11.1.2)中 $\mathbf{A}$ 被变化了的元素只是 $p$，$q$ 的行和列，写出如下形式

$$\mathbf{A}' = \begin{bmatrix} & \cdots & a'_{1p} & \cdots & a'_{1q} & \cdots & \\ \vdots & & \vdots & & \vdots & & \vdots \\ a'_{p1} & \cdots & a'_{pp} & \cdots & a'_{pq} & \cdots & a'_{pn} \\ \vdots & & \vdots & & \vdots & & \vdots \\ a'_{q1} & \cdots & a'_{qp} & \cdots & a'_{qq} & \cdots & a'_{qn} \\ \vdots & & \vdots & & \vdots & & \vdots \\ & \cdots & a'_{np} & \cdots & a'_{nq} & \cdots & \end{bmatrix} \tag{11.1.3}$$

将式(11.1.2)乘出来，并利用 $\mathbf{A}$ 的对称性，我们得到如下明确的公式

$$\left.\begin{aligned} a'_{rp} &= c a_{rp} - s\, a_{rq} \\ a'_{rq} &= c a_{rq} + s\, a_{rp} \end{aligned}\right\} \quad r \neq p, r \neq q \tag{11.1.4}$$

$$a'_{pp} = c^2 a_{pp} + s^2 a_{qq} - 2sc\,a_{pq} \tag{11.1.5}$$

$$a'_{qq} = s^2 a_{pp} + c^2 a_{qq} + 2sc\,a_{pq} \tag{11.1.6}$$

$$a'_{pq} = (c^2 - s^2) a_{pq} + sc(a_{pp} - a_{qq}) \tag{11.1.7}$$

雅可比方法的思想是，试图通过一系列平面旋转将非对角元素零化。相应地，令 $a'_{pq}=0$，从式(11.1.7)给出如下的旋转角表达式

$$\theta \equiv \cot 2\varphi \equiv \frac{c^2 - s^2}{2sc} = \frac{a_{qq} - a_{pp}}{2a_{pq}} \tag{11.1.8}$$

如果我们令 $t \equiv s/c$，$\theta$ 的定义可以重写成

$$t^2 + 2t\theta - 1 = 0 \tag{11.1.9}$$

这一方程较小的根对应着一个数量上小于 $\pi/4$ 的旋转角。每一步做这样的选择就可给出最稳定的化简。利用二次求根公式并对分母进行选择，我们可以写出这一较小的根为

$$t = \frac{\operatorname{sgn}(\theta)}{|\theta| + \sqrt{\theta^2 + 1}} \tag{11.1.10}$$

如果 $\theta$ 很大，以至于 $\theta^2$ 将会在计算机上溢出，我们令 $t = 1/(2\theta)$。于是就有

$$c = \frac{1}{\sqrt{t^2 + 1}} \tag{11.1.11}$$

$$s = tc \tag{11.1.12}$$

当我们数值地应用式(11.1.4)~(11.1.7)时，我们将它们重写，以使舍入误差达到最小。式(11.1.7)由下式代替

$$a'_{pq} = 0 \tag{11.1.13}$$

剩下公式的指导思想就是，将它们的新值写成旧值加上一个小的修正。于是我们利用式(11.1.7)和(11.1.13)将式(11.1.5)中的 $a_{qq}$ 消去，得到

$$a'_{pp} = a_{pp} - t a_{pq} \tag{11.1.14}$$

类似地有，

$$a'_{qq} = a_{qq} + t a_{pq} \tag{11.1.15}$$

$$a'_{rp} = a_{rp} - s(a_{rq} + \tau\, a_{rp}) \tag{11.1.16}$$

$$a'_{rq} = a_{rq} + s(a_{rp} - \tau\, a_{rq}) \tag{11.1.17}$$

其中 $\tau$（$=\tan\varphi/2$）定义为

$$\tau = \frac{s}{1+c} \tag{11.1.18}$$

可以通过所有非对角元素的平方和看出雅可比方法的收敛性

$$S = \sum_{r\neq s} |a_{rs}|^2 \tag{11.1.19}$$

由式(11.1.4)至(11.1.7)可得

$$S' = S - 2|a_{pq}|^2 \tag{11.1.20}$$

(因为转换是正交的，对角元素的平方和相应地增加$2|a_{pq}|^2$。)于是 $S$ 的序列单调递减。因为序列的下界为零，而且我们可选择 $a_{pq}$ 为我们所需要的值，故这一序列可以收敛到零。

最终可以得到在机器精度内的对角矩阵 **D**。它的对角元素给出原来矩阵 **A** 的特征值，由于

$$\mathbf{D} = \mathbf{V}^T \cdot \mathbf{A} \cdot \mathbf{V} \tag{11.1.21}$$

其中

$$\mathbf{V} = \mathbf{P}_1 \cdot \mathbf{P}_2 \cdot \mathbf{P}_3 \cdots \tag{11.1.22}$$

$\mathbf{P}_i$ 是一系列相继的雅可比旋转矩阵。**V** 的列是特征向量(由于 $\mathbf{A}\cdot\mathbf{V}=\mathbf{V}\cdot\mathbf{D}$)。它们可以通过每一步计算

$$\mathbf{V}' = \mathbf{V} \cdot \mathbf{P}_i \tag{11.1.23}$$

而得出，其中最开始的 **V** 是原来的矩阵 **A**。详细地，式(11.1.23)可写成

$$\begin{aligned} v'_{rs} &= v_{rs} \qquad (s \neq p, s \neq q) \\ v'_{rp} &= cv_{rp} - sv_{rq} \\ v'_{rq} &= sv_{rp} + cv_{rq} \end{aligned} \tag{11.1.24}$$

也可象式(11.1.16)和(11.1.17)那样，用 $\tau$ 将这些公式重写而减小舍入误差。

剩下的唯一问题就是消元的次序问题。1846年雅可比最初算法是，每一步搜索出矩阵的上三角部分中的最大元素，并将其置为零。这对于手算来说，是一种合理的方法；但在计算机，仅只寻找最大元素这一项就使每一步雅可比旋转的运算量是 $N^2$ 而不是 $N$。

一种更好的方法叫**循环雅可比法**，就是按严格的顺序来零化各元素。例如，可以简单地按行进行消去：$\mathbf{P}_{12},\mathbf{P}_{13},\cdots,\mathbf{P}_{1n}$；然后 $\mathbf{P}_{23},\mathbf{P}_{24}$等等。对于非退化特征值的情况，原来的方法和这种循环雅可比法一般都是二次收敛的。这样一组 $n(n-1)/2$个雅可比旋转叫做一次清扫。

下面是基于参考书[1,2]中的程序，并使用了两项改进：

- 在前三次清扫中，我们只对$|a_{pq}|>\varepsilon$ 的元素进行 $pq$ 旋转，其中 $\varepsilon$ 是某一阈值

$$\varepsilon = \frac{1}{5}\frac{S_0}{n^2} \tag{11.1.25}$$

其中 $S_0$是非对角元的模数之和

$$S_0 = \sum_{r<s} |a_{rs}| \tag{11.1.26}$$

- 四次清扫之后，如果$|a_{pq}| \ll |a_{pp}|$，并且 $|a_{pq}| \ll |a_{qq}|$，我们则令$|a_{pq}|=0$，并跳过这次旋转。比较的标准是$|a_{pq}|<10^{-(D+2)}|a_{pp}|$，其中 $D$ 是机器的十进制有效位数，对$|a_{qq}|$也同样处理。

在下列程序中，n×n 的对称矩阵 a 存储在数组 a[1..n][1..n]中。输出时 a 的超对角元

素都已被破坏;但对角和次对角元素还是原来矩阵 a 的值。向量 d[1..n]返回 a 的特征值。在计算过程中它存有当前 a 的对角元素值。矩阵 v[1..n][1..n]输出归一化特征向量,其中第 k 列对应特征值 d[k]。参数 nrot 是达到收敛所需的雅可比旋转次数。

典型的矩阵需要 6 到 10 次清扫,也就是 $3n^2$到 $5n^2$次雅可比旋转。每次旋转需要 $4n$ 次操作,每次操作包括乘和加,因而总运算理 $12n^3$到 $20n^3$量级的。计算特征值的同时若还需计算特征向量,则每次旋转的操作数从 $4n$ 变成 $6n$ 次,仅增加了 50% 的运算量。

```
#include <math.h>
#include "nrutil.h"
#define ROTATE(a,i,j,k,l) g=a[i][j];h=a[k][l];a[i][j]=g-s*(h+g*tau);\
    a[k][l]=h+s*(g-h*tau);

void jacobi(float **a, int n, float d[], float **v, int *nrot)
    计算实对称矩阵 a[1..n][1..n] 的所有特征值和特征向量。输出时 a 对角以上元素被破坏。d[1..n]返回 a 的特征
    值。输出时 v[1..n][1..n]的列是 a 的归一化特征向量。nrot 返回所需的雅可比旋转次数。
{
    int j,iq,ip,i;
    float tresh,theta,tau,t,sm,s,h,g,c,*b,*z;

    b=vector(1,n);
    z=vector(1,n);
    for (ip=1;ip<=n;ip++) {                              对单位矩阵初始赋值
        for (iq=1;iq<=n;iq++) v[ip][iq]=0.0;
        v[ip][ip]=1.0;
    }
    for (ip=1;ip<=n;ip++) {                              将 b 和 d 初始化赋值成 a 的对角元素
        b[ip]=d[ip]=a[ip][ip];
        z[ip]=0.0;                                       该向量将按(11.1.14)式中的 ta_pq
    }                                                        项积累
    *nrot=0;
    for (i=1;i<=50;i++) {
        sm=0.0;
        for (ip=1;ip<=n-1;ip++) {                        将非对角元素求和
            for (iq=ip+1;iq<=n;iq++)
                sm += fabs(a[ip][iq]);
        }
        if (sm == 0.0) {                                 正常返回, 依赖于四次收敛到机器
            free_vector(z,1,n);                              下溢出
            free_vector(b,1,n);
            return;
        }
        if (i < 4)
            tresh=0.2*sm/(n*n);                          ...在前三次扫描中
        else
            tresh=0.0;                                   ...这以后
        for (ip=1;ip<=n-1;ip++) {
            for (iq=ip+1;iq<=n;iq++) {
                g=100.0*fabs(a[ip][iq]);
                四次扫描后, 如果非对角元素很小, 则跳过旋转
                if (i > 4 && (float)(fabs(d[ip])+g) == (float)fabs(d[ip])
                    && (float)(fabs(d[iq])+g) == (float)fabs(d[iq]))
                    a[ip][iq]=0.0;
                else if (fabs(a[ip][iq]) > tresh) {
                    h=d[iq]-d[ip];
                    if ((float)(fabs(h)+g) == (float)fabs(h))
                        t=(a[ip][iq])/h;                 t = 1/(2θ)
                    else {
                        theta=0.5*h/(a[ip][iq]);         等式 (11.1.10)
                        t=1.0/(fabs(theta)+sqrt(1.0+theta*theta));
                        if (theta < 0.0) t = -t;
                    }
```

```
                    c=1.0/sqrt(1+t*t);
                    s=t*c;
                    tau=s/(1.0+c);
                    h=t*a[ip][iq];
                    z[ip] -= h;
                    z[iq] += h;
                    d[ip] -= h;
                    d[iq] += h;
                    a[ip][iq]=0.0;
                    for (j=1;j<=ip-1;j++) {          1 ≤ j < p 旋转的情形
                        ROTATE(a,j,ip,j,iq)
                    }
                    for (j=ip+1;j<=iq-1;j++) {       p < j < q 旋转的情形
                        ROTATE(a,ip,j,j,iq)
                    }
                    for (j=iq+1;j<=n;j++) {          q < j ≤ n 旋转的情形
                        ROTATE(a,ip,j,iq,j)
                    }
                    for (j=1;j<=n;j++) {
                        ROTATE(v,j,ip,j,iq)
                    }
                    ++(*nrot);
                }
            }
        }
        for (ip=1;ip<=n;ip++) {
            b[ip] += z[ip];
            d[ip]=b[ip];                             将 d 修改成 ta_pq 的和
            z[ip]=0.0;                               并重新初始化 z
        }
    }
    nrerror("Too many iterations in routine jacobi");
}
```

注意，上述程序是假设下溢被置为零。在不满足此条件的机器上，必须修改此程序。

输出时的特征值不是按顺序的。如果需要排序，可以调用下列程序将程序 **jacobi** 或本章其它程序的输出重新排序。(所用的方法是直接插入法，其运算量是 $N^2$ 而不是 $N\log N$ 量级。但由于原程序中刚刚使用了一个 $N^3$ 量级的过程来寻找特征值，所以这点开销不算大。)

```
void eigsrt(float d[], float **v, int n)
```

给定 **jacobi**(第11.1节)或 **tqli**(第11.3节)的输出特征值 d[1..n]和特征向量 v[1..n][1..n]，本程序将特征值按递减的次序，并对特征向量做相应的调整。所用方法是直接插入法。

```
{
    int k,j,i;
    float p;

    for (i=1;i<n;i++) {
        p=d[k=i];
        for (j=i+1;j<=n;j++)
            if (d[j] >= p) p=d[k=j];
        if (k != i) {
            d[k]=d[i];
            d[i]=p;
            for (j=1;j<=n;j++) {
                p=v[j][i];
                v[j][i]=v[j][k];
                v[j][k]=p;
            }
        }
    }
}
```

参考文献和进一步读物：

Smith, B. T., et 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag). [1]

Wilkinson, J. H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [2]

## 11.2 将对称矩阵简化为三对角形式：吉文斯约化和豪斯贺德约化

正如已经提到的，找寻特征值和特征向量的最优策略是，首先将矩阵化成一个简单的形式，再开始一个迭代过程。对于对称矩阵来说，最好的简单形式是三对角形式。**吉文斯约化**是对雅可比方法的一个改进。当把矩阵简化为三对角形式时我们就停止运算，而不是把它一直化到为对角形式。这就可以使整个过程用有限步完成，而不是象雅可比方法那样一直迭代到收敛。

### 11.2.1 吉文斯(Givens)方法

对于吉文斯方法，我们按式(11.1.1)选择旋转角来将某一元素零化，但这一元素不在四个"角"上，即不是式(11.1.3)中的 $a_{pp}$，$a_{pq}$和 $a_{qq}$。我们特地选择 $\mathbf{P}_{23}$来将 $a_{31}$(对称地还有 $a_{13}$)零化。然后我们选择 $\mathbf{P}_{24}$将 $a_{41}$零化。一般地，我们选择序列

$$\mathbf{P}_{23},\mathbf{P}_{24},\ldots,\mathbf{P}_{2n};\mathbf{P}_{34},\ldots,\mathbf{P}_{3n};\ldots;\mathbf{P}_{n-1,n}$$

其中 $\mathbf{P}_{jk}$用来将 $a_{k,j-1}$零化。这种方法能够奏效是由于按照式(11.1.4)$a'_{rp}$和 $a'_{rq}$($r\neq p,r\neq q$)是旧值 $a_{rp}$和 $a_{rq}$的线性组合。因而如果 $a_{rp}$和 $a_{rq}$已被零化，当约简过程继续进行时它们仍保持为零。最终需要 $n^2/2$ 次旋转，而这种方法的直接实施需要 $4n^3/3$ 量级的乘法次数，这不包括为求得特征向量而进行的矩阵乘法。

随后将要讨论的豪斯贺德方法象吉文斯约化方法一样稳定而且双倍地有效，所以通常不使用吉文斯方法。最近的工作(参见[1])表明，通过将吉文斯方法重新表述整理，可以将操作次数减少一倍，并且可以避免取平方根的运算。这使得这种算法可以和豪斯贺德约化方法相竞争。尽管这样，我们还是没有足够的经验为理由来认为这种方法比豪斯贺德方法更可取。

### 11.2.1 豪斯贺德(Householder)方法

豪斯贺德算法用 $n-2$个正交变换将 $n\times n$ 的对称矩阵 $\mathbf{A}$ 约简成三对角形式。每个变换将一整行和一整列中的相应元素零化。基本的成分是如下形式的豪斯贺德矩阵 $\mathbf{P}$

$$\mathbf{P}=\mathbf{1}-2\mathbf{w}\cdot\mathbf{w}^T \tag{11.2.1}$$

其中 $\mathbf{w}$ 是满足 $|\mathbf{w}|^2=1$的实向量。(在当前的表示中，两个向量 $\mathbf{a}$ 和 $\mathbf{b}$ 的外积或矩阵积写成 $\mathbf{a}\cdot\mathbf{b}^T$，而它们的内积或标量积写成 $\mathbf{a}^T\cdot\mathbf{b}$。)矩阵 $\mathbf{P}$ 是正交的，因为

$$\begin{aligned}\mathbf{P}^2&=(\mathbf{1}-2\mathbf{w}\cdot\mathbf{w}^T)\cdot(\mathbf{1}-2\mathbf{w}\cdot\mathbf{w}^T)\\&=\mathbf{1}-4\mathbf{w}\cdot\mathbf{w}^T+4\mathbf{w}\cdot(\mathbf{w}^T\cdot\mathbf{w})\cdot\mathbf{w}^T\\&=\mathbf{1}\end{aligned}\tag{11.2.2}$$

所以 $\mathbf{P}=\mathbf{P}^{-1}$。但因 $\mathbf{P}^T=\mathbf{P}$，于是 $\mathbf{P}^T=\mathbf{P}^{-1}$，这就证明了 $\mathbf{P}$ 的正交性。

将 $\mathbf{P}$ 重写成

$$\mathbf{P} = \mathbf{1} - \frac{\mathbf{u} \cdot \mathbf{u}^T}{H} \tag{11.2.3}$$

其中的标量 $H$ 是

$$H \equiv \frac{1}{2} |\mathbf{u}|^2 \tag{11.2.4}$$

现在 $\mathbf{u}$ 可以是任何向量。假设 $\mathbf{x}$ 是矩阵 $\mathbf{A}$ 的第一列代表的向量。选择

$$\mathbf{u} = \mathbf{x} \mp |\mathbf{x}|\mathbf{e}_1 \tag{11.2.5}$$

其中 $\mathbf{e}_1$ 是单位向量 $[1,0,\ldots,0]^T$，有关符号的选择后面将讨论。于是有

$$\begin{aligned}
\mathbf{P} \cdot \mathbf{x} &= \mathbf{x} - \frac{\mathbf{u}}{H} \cdot (\mathbf{x} \mp |\mathbf{x}|\mathbf{e}_1)^T \cdot \mathbf{x} \\
&= \mathbf{x} - \frac{2\mathbf{u} \cdot (|\mathbf{x}|^2 \mp |\mathbf{x}| x_1)}{2|\mathbf{x}|^2 \mp 2|\mathbf{x}| x_1} \\
&= \mathbf{x} - \mathbf{u} \\
&= \pm |\mathbf{x}|\mathbf{e}_1
\end{aligned} \tag{11.2.6}$$

这表明豪斯贺德矩阵 $\mathbf{P}$ 作用在给定向量 $\mathbf{x}$ 上就将零化除第一个元素外的所有元素。

为了将对称矩阵 $\mathbf{A}$ 化为三对角形式，我们选择第一个豪斯贺德矩阵的向量 $\mathbf{x}$ 为矩阵 $\mathbf{A}$ 第一列下面的 $n-1$ 个元素。于是以下 $n-2$ 个元素将被零化：

$$\mathbf{P}_1 \cdot \mathbf{A} = \left[\begin{array}{c|cccc} 1 & 0 & 0 & \cdots & 0 \\ \hline 0 & & & & \\ 0 & & & & \\ \vdots & & {}^{(n-1)}\mathbf{P}_1 & & \\ 0 & & & & \end{array}\right] \cdot \left[\begin{array}{c|cccc} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ \hline a_{21} & & & & \\ a_{31} & & & & \\ \vdots & & \text{无关部分} & & \\ a_{n1} & & & & \end{array}\right] = \left[\begin{array}{c|cccc} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ \hline k & & & & \\ 0 & & & & \\ \vdots & & \text{无关部分} & & \\ 0 & & & & \end{array}\right] \tag{11.2.7}$$

这里我们将矩阵写成分块的形式，${}^{(n-1)}\mathbf{P}$ 表示一个 $(n-1)\times(n-1)$ 维的豪斯贺德矩阵。$k$ 简单地就是向量 $[a_{21},\ldots a_{n1}]^T$ 的模的正值或负值。

完整的正交变换是

$$\mathbf{A}' = \mathbf{P} \cdot \mathbf{A} \cdot \mathbf{P} = \left[\begin{array}{c|cccc} a_{11} & k & 0 & \cdots & 0 \\ \hline k & & & & \\ 0 & & & & \\ \vdots & & \text{无关部分} & & \\ 0 & & & & \end{array}\right] \tag{11.2.8}$$

这里我们已经用到了 $\mathbf{P}^T = \mathbf{P}$。

现在选择第二个豪斯贺德矩阵的 $\mathbf{x}$ 为第二列下面的 $n-2$ 个元素，从而构成矩阵

$$\mathbf{P}_2 = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & & & \\ \vdots & \vdots & & {}^{(n-2)}\mathbf{P}_2 & \\ 0 & 0 & & & \end{bmatrix} \tag{11.2.9}$$

左上角的单位块保证了第一步已经三对角化了的部分不会被破坏，而$(n-2)$维的豪斯贺德矩阵${}^{(n-2)}\mathbf{P}_2$将新的一行和一列三对角化。显然，一系列$n-2$个这样的变换就可将矩阵 **A** 化为三对角形式。

为了避免直接做矩阵乘法 $\mathbf{P}\cdot\mathbf{A}\cdot\mathbf{P}$，我们计算一个向量

$$\mathbf{p} \equiv \frac{\mathbf{A}\cdot\mathbf{u}}{H} \tag{11.2.10}$$

则

$$\mathbf{A}\cdot\mathbf{P} = \mathbf{A}\cdot\left(1 - \frac{\mathbf{u}\cdot\mathbf{u}^T}{H}\right) = \mathbf{A} - \mathbf{p}\cdot\mathbf{u}^T$$

$$\mathbf{A}' = \mathbf{P}\cdot\mathbf{A}\cdot\mathbf{P} = \mathbf{A} - \mathbf{p}\cdot\mathbf{u}^T - \mathbf{u}\cdot\mathbf{p}^T + 2K\mathbf{u}\cdot\mathbf{u}^T$$

其中 $K$ 定义为

$$K = \frac{\mathbf{u}^T\cdot\mathbf{p}}{2H} \tag{11.2.11}$$

如果记

$$\mathbf{q} \equiv \mathbf{p} - K\mathbf{u} \tag{11.2.12}$$

就得到

$$\mathbf{A}' = \mathbf{A} - \mathbf{q}\cdot\mathbf{u}^T - \mathbf{u}\cdot\mathbf{q}^T \tag{11.2.13}$$

这是一个计算上很用的公式。

按照参考文献[2]，下面给出的豪斯贺德约化程序是从 **A** 的第 $n$ 列开始，而不是如上面介绍的从第一列开始。有关的公式如下：在第 $m$ 步$(m=1,2,\ldots,n-2)$向量 **u** 具有形式

$$\mathbf{u}^T = [a_{i1}, a_{i2}, \ldots, a_{i,i-2}, a_{i,i-1} \pm \sqrt{\sigma}, 0, \ldots, 0] \tag{11.2.14}$$

其中

$$i \equiv n - m + 1 = n, n-1, \ldots, 3 \tag{11.2.15}$$

$\sigma$(在以前的记号为$|x|^2$)是

$$\sigma = (a_{i1})^2 + \ldots + (a_{i,i-1})^2 \tag{11.2.16}$$

为了减小舍入误差，选择式(11.2.14)中的 $\sigma$ 的符号与 $a_{i,i-1}$ 的符号相同。

各变量的计算按下列顺序进行：$\sigma, \mathbf{u}, H, \mathbf{p}, K, \mathbf{q}, \mathbf{A}'$。在进行第 $m$ 步时，**A** 的最后 $m-1$ 行和列已被三对角化。

如果最后三对角矩阵的特征向量已被找到(例如，通过下一节的程序)，则 **A** 的特征向量就是将下列联合变换作用到那些特征向量上。

$$\mathbf{Q} = \mathbf{P}_1\cdot\mathbf{P}_2\ldots\mathbf{P}_{n-2} \tag{11.2.17}$$

在所有 **P** 确定之后可通过递归来得到：

$$\mathbf{Q}_{n-2} = \mathbf{P}_{n-2}$$

$$\mathbf{Q}_j = \mathbf{P}_j\cdot\mathbf{Q}_{j+1}, \qquad j = n-3, \ldots, 1$$

$$\mathbf{Q} = \mathbf{Q}_1 \tag{11.2.18}$$

下面程序的输入是实的对称矩阵 a[1..n][1..n]。输出时，$a$ 中包含有正交矩阵 q 的元素。向量 d[1..n]被置成三对角矩阵 A′ 的对角元素，而向量 e[1..n]从 e[2]到 e[n]存有 A′ 非对角元素，其中 e[1]=0。注意由于计算中 a 被重新写入数值，如果后续计算中需要的话，应该在程序调用前将 a 复制并保存起来。

不需要对中间结果设置特殊的存储数组。在第 $m$ 步，向量 **p** 和 **q** 只有1，…，$i$ 的元素是非零的(回忆一下 $i=n-m-1$)，而 u 只有第 1，…，$i-1$元素是非零的。向量 e 的元素是按照第 $n$，$n-1$，…的顺序确定的，所以可将向量 **p** 的非零元素存储在向量 e 的尚未确定部分，向量 **q** 可在 **p** 一旦不再需要时写在 **p** 的位置上。将 u 存储在 a 的第 $i$ 行，并把 u/$H$ 存储在 a 的第 $i$ 列，约化结束之后，可用存储在 a 中的 **u** 和 u/$H$ 来计算矩阵 $\mathbf{Q}_j$。因为 $\mathbf{Q}_j$ 的后 $n-j+1$行和列是单位矩阵，故只需计算前 $n-j$ 行和前 $n-j$ 列。这就可以重新存入 a 中 **u** 和 u/$H$ 的对应行和列，因为后续 **Q** 不再需要这些位置的元素。

下面给出的程序 tred2 还包括一项改进。如果在任何一步中，数量 $\sigma$ 是零或"小量"，就可以跳过对应变换。一个简单的标准，例如下式

$$\sigma < \frac{\text{在机器上可表示的最小正数}}{\text{机器精度}}$$

在大多数情况下是合适的。实际上，所应用的是一种更详细的标准。定义量

$$\varepsilon = \sum_{k=1}^{i-1} |a_{ik}| \tag{11.2.19}$$

如果在机器精度下 $\varepsilon=0$，我们就跳过这次变换。否则我们重新定义，将

$$a_{ik} \quad \text{变为} \quad a_{ik}/\varepsilon \tag{11.2.20}$$

并用重新定标了的变量进行变换。(一个豪斯贺德变换只取决于矩阵元素之间的比例。)

注意，当所处理的矩阵的元素数值大小跨跃很大数量级时，需将矩阵进行置换，要尽量置换成小的元素在矩阵的左上角，这一点很重要。因为约化是从右下角开始，而大小元素的混杂会带来很大的舍入误差。

程序 **tred2** 被设计成与下一节的程序 **tqli** 联合使用。**tqli** 是用来计算对称三对角矩阵的特征值和特征向量的。在计算实对称矩阵特征值和特征向量的技术中，**tred2** 和 **tqli** 的结合是已知最有效的方法。

在下列清单中，由注释标出的语句只有在计算特征向量时才有用。如果只需计算特征值，则在大 $n$ 值情况下，忽略这些注出的语句可以使 **tred2** 的运行速度提高一倍。在较大 $n$ 值的限制下，只计算特征值的豪斯贺德约化的运算量是 $2n^3/3$ 阶，而特征值和特征向量都计算时需用 $4n^3/3$ 阶的运算量。

```
#include <math.h>

void tred2(float **a, int n, float d[], float e[])
    对实对称矩阵 a[1..n][1..n]的豪斯贺德约化。输出时，a 被产生变换的正交矩阵 Q 取代。d[1..n]返回三对角矩阵
    的对角元素，e[1..n]返回非对角元素，且有 e[1]=0。正象注释中注明的，有些语句在只需求特征值的情况下可略
    去，这时 a 中不返回有用的信息。在其它情况下，这些语句应包含在程序中。
{
    int l,k,j,i;
    float scale,hh,h,g,f;
```

```
    for (i=n;i>=2;i--) {
        l=i-1;
        h=scale=0.0;
        if (l > 1) {
            for (k=1;k<=l;k++)
                scale += fabs(a[i][k]);
            if (scale == 0.0)                    跳过变换
                e[i]=a[i][l];
            else {
                for (k=1;k<=l;k++) {
                    a[i][k] /= scale;            将标定的a用于变换
                    h += a[i][k]*a[i][k];        在h中形成σ
                }
                f=a[i][l];
                g=(f >= 0.0 ? -sqrt(h) : sqrt(h));
                e[i]=scale*g;
                h -= f*g;                        现在h是(11.2.4)式
                a[i][l]=f-g;                     将u存入a的第i行
                f=0.0;
                for (j=1;j<=l;j++) {
                /* Next statement can be omitted if eigenvectors not wanted */
                    a[j][i]=a[i][j]/h;           将u/H存入a的第i列
                    g=0.0;                       在g中形成A·u的一个元素
                    for (k=1;k<=j;k++)
                        g += a[j][k]*a[i][k];
                    for (k=j+1;k<=l;k++)
                        g += a[k][j]*a[i][k];
                    e[j]=g/h;                    在e的暂时不用的元素中形成P的元素

                    f += e[j]*a[i][j];
                }
                hh=f/(h+h);                      形成K,(11.2.11)式
                for (j=1;j<=l;j++) {             形成q并存入e中p的位置上
                    f=a[i][j];                   注意有e[l]=e[i-1]
                    e[j]=g=e[j]-hh*f;
                    for (k=1;k<=j;k++)           约化a,(11.2.13)式
                        a[j][k] -= (f*e[k]+g*a[i][k]);
                }
            }
        } else
            e[i]=a[i][l];
        d[i]=h;
    }
    /* Next statement can be omitted if eigenvectors not wanted */
    d[1]=0.0;
    e[1]=0.0;
    /* Contents of this loop can be omitted if eigenvectors not
            wanted except for statement d[i]=a[i][i]; */
    for (i=1;i<=n;i++) {                         开始矩阵变换的积累
        l=i-1;
        if (d[i]) {                              当i=1时跳过这一块
            for (j=1;j<=l;j++) {
                g=0.0;
                for (k=1;k<=l;k++)               利用a中存贮的u和u/H来形成P·Q
                    g += a[i][k]*a[k][j];
                for (k=1;k<=l;k++)
                    a[k][j] -= g*a[k][i];
            }
        }
        d[i]=a[i][i];                            这一句保留
        a[i][i]=1.0;                             为下一次迭代将a重新置成单位矩阵
        for (j=1;j<=l;j++) a[j][i]=a[i][j]=0.0;
    }
}
```

参考文献和进一步读物：

Golub, G. H., and Van Loan, C. F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §5.1. [1]

Wilkinson, J. H., and Reinsch, C. 1971, *Linear Algebra*, vol. Ⅱ of *Handbook for Automatic Computation* (New York: Springer-Verlag). [2]

## 11.3 三对角矩阵的特征值和特征向量

### 11.3.1 特征多项式的赋值

一旦把原先的实对称矩阵约化为三对角矩阵，确定它的特征值的一种可能的途径就是直接寻找特征值的多项式 $p_n(\lambda)$的根。对于任何试验值 $\lambda$，三对角矩阵的特征多项式都可以通过一种有效的递归关系进行求值(例如，参[1])。递归过程中产生的较低阶多项式形成一个斯特梅恩(Sturmian)序列，该序列可用来将特征值定位于实轴上的某些区间。然后可以进一步使用象二分法或牛顿法那样的寻根方法来更精确地确定区间。相应的特征向量可以通过逆迭代找到。

基于上述思想的程序可以在参考文献[2,3]中找到。然而，如果只需计算一小部分特征值和特征向量，则下面讨论的因子分解的方法更加有效。

### 11.3.2 *QR* 和 *QL* 算法

*QR* 算法的基本思想是，任何实矩阵可以分解成下列形式

$$\mathbf{A} = \mathbf{Q} \cdot \mathbf{R} \tag{11.3.1}$$

其中 **Q** 是正交的，而 **R** 是上三角的。对于一般矩阵，这种分解可以通过采用豪斯贺德变换，逐列地将矩阵 **A** 的对角线以下部分零化来构成。

现在，考虑将式(11.3.1)中因子交换所得到的矩阵：

$$\mathbf{A}' = \mathbf{R} \cdot \mathbf{Q} \tag{11.3.2}$$

因为 **Q** 是正交的，式(11.3.1)给出 $\mathbf{R}=\mathbf{Q}^T\cdot\mathbf{A}$。所以(11.3.2)式变成

$$\mathbf{A}' = \mathbf{Q}^T \cdot \mathbf{A} \cdot \mathbf{Q} \tag{11.3.3}$$

我们看到 **A**′ 和 **A** 的一个正交变换。

可以验证，*QR* 变换保持了矩阵的下列性质：对称性、三对角形式和海森伯格(Hesserberg)形式(将第11.5节中定义)。

将 **A** 的因子之一选为上三角矩阵没有特别的意义。也可以等价地使它是下三角形式，这就叫做 *QL* 算法，因为

$$\mathbf{A} = \mathbf{Q} \cdot \mathbf{L} \tag{11.3.4}$$

其中 **L** 是下三角的。(这种标准的，但又使人迷惑的术语 *R* 和 *L* 代表矩阵的右边或左边为非零。)

回忆一下，在第11.2节中豪斯贺德方法将矩阵三对角化时，我们是从原来矩阵的第 $n$(最后)列开始的。为减小舍入误差，我们那时建议尽量将矩阵的最大元素放到右下角。如果我们现在希望将所得到的三对角矩阵对角化，则 *QL* 算法比 *QR* 算法的舍入误差小，所以今后我们使用 *QL* 算法。

$QL$ 算法包括一系列正交变换：

$$
\begin{aligned}
\mathbf{A}_s &= \mathbf{Q}_s \cdot \mathbf{L}_s \\
\mathbf{A}_{s+1} &= \mathbf{L}_s \cdot \mathbf{Q}_s \quad (= \mathbf{Q}_s^T \cdot \mathbf{A}_s \cdot \mathbf{Q}_s)
\end{aligned}
\tag{11.3.5}
$$

下列定理(不是那么显然!)是关于一般矩阵 **A** 的算法的基础：(i)如果 **A** 的特征值具有不同的绝对值 $|\lambda_i|$，则当 $s \to \infty$ 时 $\mathbf{A}_s \to$ [下三角形式]。对角线上出现的特征值按某绝对值递增的顺序排列。(ii)如果 **A** 有 $p$ 重特征值 $|\lambda_i|$，则当 $s \to \infty$ 时 $\mathbf{A}_s \to$ [下三角形式]，但除去一个特征值$\to\lambda_i$的 $p$ 维对角子块阵。这一定理的证明篇幅很长，可参见[4]。

$QL$ 算法对于一般矩阵的运算量是每次迭代 $O(n^3)$，这对于一般矩阵来说运算量太大了。但它对于三对角矩阵每次迭代的运算量只有 $O(n)$，而对于海森柏格矩阵是 $O(n^2)$，故使这算法用于这两种形式的矩阵是很有效的。

在这一节中，我们只考虑 **A** 是实、对称的三对角矩阵，因而所有的特征值$\lambda_i$是实的。根据上述定理，如果任一 $\lambda_i$是 $p$ 重的，则在次对角和超对角上至少有 $p-1$个零元。于是，矩阵可以分裂成能够单独对角化的子矩阵，而且在一般情形下，由对角块带来复杂性就无关紧要了。

在上述定理的证明过程中，人们发现，一般超对角元素按下列方式收敛于零

$$
a_{ij}^{(s)} \sim \left(\frac{\lambda_i}{\lambda_j}\right)^s \tag{11.3.6}
$$

虽然 $\lambda_i<\lambda_j$，但如果 $\lambda_i$与 $\lambda_j$很接近的话，收敛的速度可能很慢。收敛可以通过**位移**的技巧来加速：如果 $k$ 是任意常数，则 $\mathbf{A}-k\mathbf{1}$具有特征值 $\lambda_i-k$ 。如果我们分解

$$
\mathbf{A}_s - k_s\mathbf{1} = \mathbf{Q}_s \cdot \mathbf{L}_s \tag{11.3.7}
$$

于是

$$
\begin{aligned}
\mathbf{A}_{s+1} &= \mathbf{L}_s \cdot \mathbf{Q}_s + k_s\mathbf{1} \\
&= \mathbf{Q}_s^T \cdot \mathbf{A}_s \cdot \mathbf{Q}_s
\end{aligned}
\tag{11.3.8}
$$

则收敛由以下比例确定

$$
\frac{\lambda_i - k_s}{\lambda_j - k_s} \tag{11.3.9}
$$

这种思想是要在每一步选择位移量 $k_s$，以使收敛的速度最快。对于位移量的一种好的初始选择是使 $k_s$ 接近最小的特征值 $\lambda_1$。于是第一列的非对角元素将很快趋于零。可是 $\lambda_1$通常事先并非是已知的。在实践中，一个很有效的方法(虽然没有证明它是最优的)是，计算 **A** 的首2×2对角子矩阵，然后令 $k_s$ 等于离 $a_{11}$最近的那个特征值。

更一般地，假设已经找到 **A** 的 $r-1$个特征值。则可以删去前 $r-1$个行和列而将矩阵简化成

$$
\mathbf{A} = \begin{bmatrix}
0 & & \cdots & \cdots & & & 0 \\
 & \cdots & & & & & \\
 & & 0 & & & & \\
\vdots & & & d_r & e_r & & \vdots \\
\vdots & & & e_r & d_{r+1} & & \\
 & & & & & \vdots & 0 \\
 & & & & & d_{n-1} & e_{n-1} \\
0 & & \cdots & & 0 & e_{n-1} & d_n
\end{bmatrix} \tag{11.3.10}
$$

选择 $k_s$ 等于前面2×2子矩阵的特征值中较接近 $d_r$ 的那一个。人们可以证明采用这种方法

后，算法的收敛性一般是三次方的（在退化特征值的情况下，最坏也是二次收敛的）。这样快的收敛速度使这种算法很具有吸引力。

注意经过位移后，特征值不再按绝对值递增的顺序出现在对角线上。若需要的话，可调用程序 **eigsrt**（第11.1节）。

正如前面提到的，一般矩阵的 $QL$ 分解由一系列豪斯贺德变换来完成。对于三对角矩阵来说，用平面旋转 $\mathbf{P}_{pq}$ 更加有效。使用变换序列 $\mathbf{P}_{12}, \mathbf{P}_{23}, \ldots, \mathbf{P}_{n-1,n}$ 将元素 $a_{12}, a_{23}, \ldots, a_{n-1,n}$ 零化。对称地，次对角元素 $a_{21}, a_{32}, \ldots, a_{n,n-1}$ 也被零化。因此，$\mathbf{Q}_s$ 是这些平面旋转的乘积：

$$\mathbf{Q}_s^T = \mathbf{P}_1^{(s)} \cdot \mathbf{P}_2^{(s)} \ldots \mathbf{P}_{n-1}^{(s)} \tag{11.3.11}$$

其中 $\mathbf{P}_i$ 将 $a_{i,i+1}$ 零化，注意式(11.3.11)中的不是 $\mathbf{Q}$ 而是 $\mathbf{Q}^T$，因为我们曾定义 $\mathbf{L} = \mathbf{Q}^T \cdot \mathbf{A}$。

## 11.3.3 具有隐含位移的 QL 算法

迄今为止，所描述的算法都是很成功的。然而，如果矩阵 $\mathbf{A}$ 的元素在数量级上相差很多，则从对角元中减去一个很大的 $k_s$ 会导致小的特征值损失精度。但是，隐含位移的 $QL$ 算法避免了这一问题。

隐含位移的 $QL$ 算法在数学上等价于原来的 $QL$ 算法，只是计算中不需从 $\mathbf{A}$ 中将 $k_s\mathbf{1}$ 减去。

这种算法基于下述引理：如果 $\mathbf{A}$ 是一个对称非奇异矩阵，并且 $\mathbf{B} = \mathbf{Q}^T \cdot \mathbf{A} \cdot \mathbf{Q}$，其中 $\mathbf{Q}$ 是正交的，而 $\mathbf{B}$ 是具有正的非对角元素的三对角矩阵，则只要 $\mathbf{Q}^T$ 的最后一行给出后，$\mathbf{Q}$ 和 $\mathbf{B}$ 就能完全确定。证明：令 $\mathbf{q}_i^T$ 代表矩阵 $\mathbf{Q}^T$ 的第 $i$ 行向量，则 $\mathbf{q}_i$ 就是矩阵 $\mathbf{Q}$ 的第 $i$ 列向量。关系式 $\mathbf{B} \cdot \mathbf{Q}^T = \mathbf{Q}^T \cdot \mathbf{A}$ 可表示成：

$$\begin{bmatrix} \beta_1 & \gamma_1 & & & \\ \alpha_2 & \beta_2 & \gamma_2 & & \\ & & \vdots & & \\ & & \alpha_{n-1} & \beta_{n-1} & \gamma_{n-1} \\ & & & \alpha_n & \beta_n \end{bmatrix} \cdot \begin{bmatrix} \mathbf{q}_1^T \\ \mathbf{q}_2^T \\ \vdots \\ \mathbf{q}_{n-1}^T \\ \mathbf{q}_n^T \end{bmatrix} = \begin{bmatrix} \mathbf{q}_1^T \\ \mathbf{q}_2^T \\ \vdots \\ \mathbf{q}_{n-1}^T \\ \mathbf{q}_n^T \end{bmatrix} \cdot \mathbf{A} \tag{11.3.12}$$

上述矩阵方程的第 $n$ 行是

$$\alpha_n \mathbf{q}_{n-1}^T + \beta_n \mathbf{q}_n^T = \mathbf{q}_n^T \cdot \mathbf{A} \tag{11.3.13}$$

因为 $Q$ 是正交的

$$\mathbf{q}_n^T \cdot \mathbf{q}_m = \delta_{nm} \tag{11.3.14}$$

如果将式(11.3.13)两边右乘 $\mathbf{q}_n$，则发现

$$\beta_n = \mathbf{q}_n^T \cdot \mathbf{A} \cdot \mathbf{q}_n \tag{11.3.15}$$

由于 $\mathbf{q}_n$ 已知，故上式是已知的。再由式(11.3.13)给出

$$\alpha_n \mathbf{q}_{n-1}^T = \mathbf{z}_{n-1}^T \tag{11.3.16}$$

其中

$$\mathbf{z}_{n-1}^T \equiv \mathbf{q}_n^T \cdot \mathbf{A} - \beta_n \mathbf{q}_n^T \tag{11.3.17}$$

是已知的。所以

$$\alpha_n^2 = \mathbf{z}_{n-1}^T \mathbf{z}_{n-1}, \tag{11.3.18}$$

或者

$$\alpha_n = |\mathbf{z}_{n-1}| \tag{11.3.19}$$

并且

$$q_{n-1}^T = z_{n-1}^T / \alpha_n \qquad (11.3.20)$$

(其中假设$\alpha_n$不为零)。类似地,可以推证出如果已知 $q_n, q_{n-1}, \ldots, q_{n-j}$和脚标大于 $n-j$ 的 $\alpha$、$\beta$ 和 $\gamma$,就可以知道脚标为 $n-(j-1)$的有关量。

将上述引理用于实际计算,假设通过某种方法找到了一个三对角矩阵 $\overline{A}_{s+1}$满足

$$\overline{A}_{s+1} = \overline{Q}_s^T \cdot \overline{A}_s \cdot \overline{Q}_s \qquad (11.3.21)$$

其中 $\overline{Q}_s^T$ 是正交的并与原来 $QL$ 算法中的 $Q_s^T$ 有相同的最后的一行。则有 $\overline{Q}_s = Q_s$ 和 $\overline{A}_{s+1} = A_{s+1}$。

在原先的算法中,从式(11.3.11)我们看到 $Q_s^T$ 的最后一行与 $P_{n-1}^{(s)}$ 的最后一行相同。回忆一下,$P_{n-1}^{(s)}$是一个用来将 $A_s - k_s 1$的第$(n-1,n)$元素零化的平面旋转。

从式(11.1.1)出发进行简单的计算给出它的参数

$$c = \frac{d_n - k_s}{\sqrt{e_n^2 + (d_n - k_s)^2}}, \quad s = \frac{-e_{n-1}}{\sqrt{e_n^2 + (d_n - k_s)^2}} \qquad (11.3.22)$$

矩阵 $P_{n-1}^{(s)} \cdot A_s \cdot P_{n-1}^{(s)T}$ 除去两个多余元素外是三对角的:

$$\begin{bmatrix} \cdots & & & & & \\ & \times & \times & \times & & \\ & & \times & \times & \times & \mathbf{x} \\ & & & \times & \times & \times \\ & & & \mathbf{x} & \times & \times \end{bmatrix} \qquad (11.3.23)$$

现在必须使用一个最后一行为$[0,0,\ldots,1]$的正交矩阵将上述矩阵三对角化,这样 $\overline{Q}_s^T$ 的最后一行就和 $P_{n-1}^{(s)}$的最后一行相同,这可以通过一系列豪斯贺德或吉文斯变换来完成。对于式(11.3.23)这样特殊形式的矩阵来说,吉文斯方法更好。我们在平面$(n-2,n-1)$中进行旋转使元素$(n-2,n)$零化。(对称地,元素$(n,n-2)$也将被清零。)这样便得到一个除去多余元素$(n-3)(n-1)$和$(n-3,n-2)$外都已三对角化了的矩阵。我们再由在平面$(n-3,n-2)$上的旋转零化这两元素,如此类推。于是我们需要 $n-2$个吉文斯旋转的序列。这就导致

$$Q_s^T = \overline{Q}_s^T = \overline{P}_1^{(s)} \cdot \overline{P}_2^{(s)} \ldots \overline{P}_{n-2}^{(s)} \cdot P_{n-1}^{(s)} \qquad (11.3.24)$$

其中 $\overline{P}$ 是吉文斯旋转,而 $P_{n-1}$是原先算法中那种平面旋转。式(11.3.21)给出了下一次迭代所需的 A。注意,位移量 $k_s$ 已经隐含在公式式(11.3.22)中了。

下面基于文献[2,3]的程序 **tqli**("三对角形 $QL$ 隐含位移方法"),在实践中运行良好,对于前几个特征值的迭代次数可能是 4 或 5 次,但同时右下角的非对角元素也会相应地减少。后面的特征值就只需很少的计算量就可求出。每个特征值的平均迭代次数一般为 1.3~1.6 次。每次迭代的操作数为 $O(n)$,并有很高的有效系数$(\sim 20n)$。因此,对角化所需的操作总数为$\sim 20n \times (1.3\sim 1.6)n \sim 30n^2$。如果同时需计算特征向量,则注释中标注出的语句需包含在程序中,而且将增加高达 $3n^3$的运算量。

```
#include <math.h>
#include "nrutil.h"

void tqli(float d[], float e[], int n, float **z)
```

为确定实对称三对角矩阵,或者事先由第11.2节中程序 tred2简化了的实对称矩阵的特征值和特征向量,本程序采用的是隐含位移的 $QL$ 算法。输入时,d[1..n]中存有三对角矩阵的对角元素,输出时它返回特征值,向量 e[1..n]输入三对角矩阵的次对角元素,e[1]为任意值,输出时,e 中的值已被破坏。如果只需求特征值,程序中有几行由注释标明的语句可以略去。如果需要求一个三对角矩阵的特征向量,则矩阵 z[1..n][1..n]中输入单位矩阵;如果需要求一个由程序 tred2已约化的矩阵之特征向量,则 z 由 tred2输出的矩阵作为输入。不管是在哪种情况下,z 的第 k 列返回与 d[k]相对应的归一化特征向量。

```
{
    float pythag(float a, float b);
    int m,l,iter,i,k;
    float s,r,p,g,f,dd,c,b;

    for (i=2;i<=n;i++) e[i-1]=e[i];                 为方便重编号 e 中元素
    e[n]=0.0;
    for (l=1;l<=n;l++) {
        iter=0;
        do {
            for (m=l;m<=n-1;m++) {                   寻找一个单一的小的次对角元素
                dd=fabs(d[m])+fabs(d[m+1]);             用来分裂矩阵
                if ((float)(fabs(e[m])+dd) == dd) break;
            }
            if (m != l) {
                if (iter++ == 30) nrerror("Too many iterations in tqli");
                g=(d[l+1]-d[l])/(2.0*e[l]);           形成位移
                r=pythag(g,1.0);
                g=d[m]-d[l]+e[l]/(g+SIGN(r,g));         这是 dm - ks.
                s=c=1.0;
                p=0.0;
                for (i=m-1;i>=l;i--) {                如同原来QL方法一样的平面转动,
                    f=s*e[i];                           随后吉文斯旋转, 用以恢复三
                    b=c*e[i];                           对角形式
                    e[i+1]=(r=pythag(f,g));
                    if (r == 0.0) {                   从下溢处恢复
                        d[i+1] -= p;
                        e[m]=0.0;
                        break;
                    }
                    s=f/r;
                    c=g/r;
                    g=d[i+1]-p;
                    r=(d[i]-g)*s+2.0*c*b;
                    d[i+1]=g+(p=s*r);
                    g=c*r-b;
                    /* Next loop can be omitted if eigenvectors not wanted*/
                    for (k=1;k<=n;k++) {              形成特征向量
                        f=z[k][i+1];
                        z[k][i+1]=s*z[k][i]+c*f;
                        z[k][i]=c*z[k][i]-s*f;
                    }
                }
                if (r == 0.0 && i >= l) continue;
                d[l] -= p;
                e[l]=g;
                e[m]=0.0;
            }
        } while (m != l);
    }
}
```

参考文献和进一步读物：


Acton, F. S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), pp. 331～335. [1]

Wilkinson, J. H., and Reinsch, C. 1971, *Linear Algebra*, vol. Ⅱ of *Handbook for Automatic Computation* (New York: Springer-Verlag). [2]

Smith, B. T., et al. 1976, *Matrix Eigensystem Routines—EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag). [3]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §6.6.6. [4]

## 11.4 埃尔米特矩阵

在复域中，实对称矩阵所类似的是埃尔米特(Hermite)矩阵，满足式(11.0.4)。雅可比变换可以用来寻找它的特征值和特征向量，如同用豪斯贺交换将它约化为三对角形式以及使用 $QL$ 迭代一样。以前的程序 **jacobi**，**tred2**和 **tqli** 在复域中的形式与在实域中极为类似，具体的工作程序，参见[1,2]。

一种替代的方法是本书中应用的程序，是将埃尔米特问题化成实对称问题：如果 $\mathbf{C}=\mathbf{A}+i\mathbf{B}$ 是一个埃尔米特矩阵，则 $n\times n$ 复特征值问题为

$$(\mathbf{A}+i\mathbf{B})\cdot(\mathbf{u}+i\mathbf{v})=\lambda(\mathbf{u}+i\mathbf{v}) \tag{11.4.1}$$

等价于一个 $2n\times 2n$ 的实问题

$$\begin{bmatrix}\mathbf{A} & -\mathbf{B}\\ \mathbf{B} & \mathbf{A}\end{bmatrix}\cdot\begin{bmatrix}\mathbf{u}\\ \mathbf{v}\end{bmatrix}=\lambda\begin{bmatrix}\mathbf{u}\\ \mathbf{v}\end{bmatrix} \tag{11.4.2}$$

注意式(11.4.2)中的 $2n\times 2n$ 矩阵是对称的：如果 $\mathbf{C}$ 是埃尔米特的，则 $\mathbf{A}^T=\mathbf{A}$，并且 $\mathbf{B}^T=-\mathbf{B}$。

对应于一个给定的特征值 $\lambda$，向量

$$\begin{bmatrix}-\mathbf{v}\\ \mathbf{u}\end{bmatrix} \tag{11.4.3}$$

也是一个特征向量，它可以通过将(11.4.2)中两个蕴含的矩阵方程写开来验证这一点。因而，如果 $\lambda_1,\lambda_2,\ldots,\lambda_n$ 是 $\mathbf{C}$ 的特征值，则扩展问题(11.4.2)的 $2n$ 个特征值是 $\lambda_1,\lambda_1,\lambda_2,\lambda_2,\ldots,\lambda_n,\lambda_n$ 也就是说，每个重复两次。特征向量是 $\mathbf{u}+i\mathbf{v}$ 和 $i(\mathbf{u}+i\mathbf{v})$ 的成对形式，即它们只相差一个非实质性的相位。因而求解扩展问题(11.4.2)，并挑选一个特征值和一对特征向量，就得到了原来矩阵 $\mathbf{C}$ 的特征值和特征向量。

计算这个扩展的矩阵比原来的复矩阵运算需要多一倍的存储量。从原则上说，在计算时间上用复数算法比用扩展问题求解的效率要高一倍。

参考文献和进一步读物：


Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [1]

Smith, B. T., et al. 1976, *Matrix Eigensystem Routines—EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag). [2]


## 11.5 将一般矩阵化为海森伯格形式

在前几节中，所给出的对称矩阵的算法在实践中很令人满意。相反地，对于非对称矩阵就不可能找到同样令人满意的算法。这有两方面原因。首先，非对称矩阵的特征值可能对矩阵元素的微小变化很敏感；其次，矩阵本身可能是缺损的，因而找不到一套完备的特征向量。我们强调的这些困难是任何数值计算方法都不能“医治”的，它是某些非对称矩阵的内在性质。我们最大的希望就是，算法过程不要使这类问题的症状加剧。

舍入误差的出现只能使情况变得更糟。在有限的精度下，人们甚至于不能找到一种简便

的方法来判定一个给定矩阵究竟是否缺损。所以当前的算法一般都是试着去发现一组完备的特征向量，至于结果怎样，得由用户自己判别。如果有两个特征向量近乎平行，则该矩阵有可能缺损。

除去介绍有关文献和搜集于[1,2]中的程序外，我们将把特征向量的问题放在一边，而只给出特征值的算法。如果只需要一小部分特征向量，则可参阅第11.7节并考虑用逆迭代的方法来找寻它们。找寻非对称矩阵的全部特征向量的问题已超出了本书的范围。

### 11.5.1 配平

在进行某种算法时，特征值对舍入误差的敏感程度可以通过配平的过程来减小。若数值过程导致的特征系统的误差一般是与矩阵的欧几里得(Euclidean)范数成正比，也就是说，与所有元素的平方和之平方根成正比。配平的思想就是，用相似变换将矩阵对应的行和列的范数变得相接近，从而在不改变特征值的前提下使矩阵的总范数减小。对称矩阵是一个已经配平了的矩阵。

配平是一个 $N^2$ 次运算量的过程。所以下述程序 **balanc** 的运行时间永远不会多于找特征值所需时间百分之几。因而建议读者总要先将非对称矩阵配平。这样只会大大提高那些极度不平衡矩阵的特征值计算的精度。

正如[1]中讨论的，实际的算法是基于奥斯伯恩(Osborne)。它由对角矩阵 **D** 完成的一系列相似变换组成。为了避免在配平过程中引入舍入误差，**D** 的元素限制为用于浮点算术的基底的适当幂次(例如，对大多数机器基数是 2，但对于 IBM 主机体系结构的基数是 16 )。输出是一个对于某种范数已经配平的矩阵。这种范数是取所有矩阵元素的绝对值之和，它比用欧几里得范数更方便，而且同样有效：一种范数能有大的约化蕴含着另一种范数也有大的约化。

注意，如果矩阵的一行或一列的非对角元素全部为零，则它的对角元素就是一个特征值。如果这个特征值恰巧是很病态的(对于矩阵元素的微小变化很敏感)，那么当用程序 **hqr** (第11.6节)确定它时，将会带来相当大的误差。如果我们事先检查整个矩阵，就能确定这些孤立的特征值，并将相应的行和列从矩阵中去掉。读者应当考虑这样一种事先检查是否在应用中有用处。(对于对称矩阵，我们给出的程序在任何情况下都能准确地确定孤立的本征值。)

程序 **balanc** 并不记录对原来矩阵所进行的相似变换本身，因为我们只关心寻找非对称矩阵的特征值而不是特征向量。如果想要跟踪这些变换的话，请参阅参考文献[1~3]。

```
#include <math.h>
#define RADIX 2.0

void balanc (flaot * * a, int n)
    给定一个矩阵a[1..n][1..n]，本程序用一个具有相同特征值的配平了的矩阵取代它。而对称矩阵是已经配平的，
    故本程序对它不起作用。参数 RADIX 是机器的浮点基数。
{
    int last,j,i;
    float s,r,g,f,c,sqrdx;

    sqrdx=RADIX*RADIX;
    last=0;
    while (last == 0) {
```

```
        last=1;
        for (i=1;i<=n;i++) {                计算行和列的范数
            r=c=0.0;
            for (j=1;j<=n;j++)
                if (j != i) {
                    c += fabs(a[j][i]);
                    r += fabs(a[i][j]);
                }
            if (c && r) {                   若两者都为非零
                g=r/RADIX;
                f=1.0;
                s=c+r;
                while (c<g) {               寻找机器基数的整数幂，使它接近配平的矩阵
                    f *= RADIX;
                    c *= sqrdx;
                }
                g=r*RADIX;
                while (c>g) {
                    f /= RADIX;
                    c /= sqrdx;
                }
                if ((c+r)/f < 0.95*s) {
                    last=0;
                    g=1.0/f;
                    for (j=1;j<=n;j++) a[i][j] *= g;     采用类似的变换
                    for (j=1;j<=n;j++) a[j][i] *= f;
                }
            }
        }
    }
}
```

## 11.5.2 约化成海森伯格形式

找寻一般矩阵的特征系统的方法与对称情形是平行的。首先，我们将矩阵的约化成一种简单的形式，然后我们对这种简单的形式进行某种迭代。这里我们所用的简单形式为**海森伯格**(Hessenberg)形式。一个**上海森伯格**矩阵在其对角线下，除去第一个次对角线上元素之外全部为零。例如，对于6×6的情形，非零元素是：

$$\begin{bmatrix} \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ & \times & \times & \times & \times & \times \\ & & \times & \times & \times & \times \\ & & & \times & \times & \times \\ & & & & \times & \times \end{bmatrix}$$

至此，应该能够一眼看出，这样一种形式可以通过一系列豪斯贺德变换得到，每一变换将矩阵一列中的相应元素零化。豪斯德贺方法的确是一种可接受的方法。然而，一种替换的方法是，类似高斯选主元的消去法的过程。我们选用这种消去法，是因为它比豪斯贺德方法效率高一倍，而且我们也想向读者介绍这种方法。可以构造出这样一种矩阵，它对于豪斯贺德约化 的正交变换是稳定的，而对于消去法是不稳定的，但这种矩阵在实践中是极为罕见的。

直接高斯消去法不是一种矩阵的相似变换。因此，实际所用的消去过程有点不同，在第 $r$ 步开始之前，原来的矩阵 $\mathbf{A}\equiv\mathbf{A}_1$ 已经变成 $\mathbf{A}_r$，它的前 $r-1$ 行和 $r-1$ 列已经化为上海森伯格形式。第 $r$ 步由下列操作构成：

· 找出第 $r$ 列中对角元以下的元素中最大的一个。如果它为零则跳过以下两项操作，这

一步就完成了。否则就假设最大元素在第 $r'$ 行。

- 将第 $r'$ 行和第 $r+1$ 行相交换。这是一个选主元过程。为了使这种置换成为相似变换，同样交换第 $r'$ 列和第 $r+1$ 列。
- 对 $i=r+2,r+3,...,N$，计算乘数

$$n_{i,r+1} \equiv \frac{a_{ir}}{a_{r+1,r}}$$

从第 $i$ 行中减去 $n_{i,r+1}$ 乘以第 $r+1$ 行。

为了使这种消去成为相似变换，同样将 $n_{i,r+1}$ 乘以第 $i$ 列加到第 $r+1$ 列上去。一共需 $N-2$ 个这样步骤。

如果矩阵元素的大小数量级相差很多，则应试着重新安排矩阵，以使最大元素置于左上角。这就可以减小舍入误差，因为整个过程是从左向右进行的。

因为我们只关心特征值，程序 **elmhes** 不跟踪记录所进行的相似变换。对于较大的 $N$，运算量大约为 $5N^3/6$ 。

```
#include <math.h>
#define SWAP(g,h) {y=(g);(g)=(h);(h)=y;}

void elmhes (float **a, int n)
    用消去法将矩阵约化成海森伯格形式。实的 n×n 非对称矩阵 a[1..n][1..n]由具有相同特征值的上海森伯格矩阵
    所代替。建议(但不必须)运行本程序前先调用 balanc。输出时，海森伯格矩阵在 a[i,j]中(i≤j+1)。具有 i>j+1的元
    素应看作为零，但返回一些无用的随机数。
{
    int m,j,i;
    float y,x;

    for (m=2;m<n;m++) {                          m 在正文中称为 r + 1
        x=0.0;
        i=m;
        for (j=m;j<=n;j++) {                      寻找主元
            if (fabs(a[j][m-1]) > fabs(x)) {
                x=a[j][m-1];
                i=j;
            }
        }
        if (i != m) {                             交换行和列
            for (j=m-1;j<=n;j++) SWAP(a[i][j],a[m][j])
            for (j=1;j<=n;j++) SWAP(a[j][i],a[j][m])
        }
        if (x) {                                  执行消去
            for (i=m+1;i<=n;i++) {
                if ((y=a[i][m-1]) != 0.0) {
                    y /= x;
                    a[i][m-1]=y;
                    for (j=m;j<=n;j++)
                        a[i][j] -= y*a[m][j];
                    for (j=1;j<=n;j++)
                        a[j][m] += y*a[j][i];
                }
            }
        }
    }
}
```

参考文献和进一步读物：

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. Ⅱ of *Handbook for Automatic Computation* (New York: Springer-Verlag). [1]

Smith, B. T., et al. 1976, *Matrix Eigensystem Routines—EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag). [2]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §6.5.4. [3]

## 11.6 实海森伯格矩阵的 $QR$ 算法

回忆一下带有位移的 $QR$ 算法的下列关系式：

$$\mathbf{Q}_s\cdot(\mathbf{A}_s-k_s\mathbf{1})=\mathbf{R}_s \tag{11.6.1}$$

其中 $\mathbf{Q}$ 是正交矩阵，而 $\mathbf{R}$ 是上三角矩阵，并且有

$$\begin{aligned}\mathbf{A}_{s+1}&=\mathbf{R}_s\cdot\mathbf{Q}_s^T+k_s\mathbf{1}\\&=\mathbf{Q}_s\cdot\mathbf{A}_s\cdot\mathbf{Q}_s^T\end{aligned} \tag{11.6.2}$$

$QR$ 变换保持原来矩阵 $\mathbf{A}\equiv\mathbf{A}_1$ 的上海森伯格形式，并且每次迭代的运算量是 $O(n^2)$，而对于一般矩阵来说，却是 $O(n^3)$。当 $s\to\infty$，$\mathbf{A}_s$ 收敛于这样一种形式，在这种形式中特征值或者在对角线上被分离出来，或者是对角线上一个 $2\times2$ 子矩阵的特征值。

正如我们在第11.3节中指出的，位移是快速收敛的关键。这里一个关键的区别是非对称实矩阵可能有复特征值。这就意味着位移量 $k_s$ 可能是复数，显然需要使用复算法。

尽管如此，可以通过一个聪明的技巧来避免复算法。这一技巧依赖于一个类似于第11.3节中用于隐含位移那样的引理。我们现在需要的引理可陈述为：如果 B 是一个非奇异矩阵，满足

$$\mathbf{B}\cdot\mathbf{Q}=\mathbf{Q}\cdot\mathbf{H} \tag{11.6.3}$$

其中 $\mathbf{Q}$ 是正交矩阵，而 $\mathbf{H}$ 是上海森伯格的，则 $\mathbf{Q}$ 和 $\mathbf{H}$ 由 $\mathbf{Q}$ 的第一列完全确定。(如果 $\mathbf{H}$ 具有正的下次对角元素，则这种确定是唯一的。)这一引理可以用类似于第11.3节中对三对角矩阵所做的推证来证明。

这一引理在实践中的应用是采用两步 $QR$ 算法，或者取两个实的位移量 $k_s$ 和 $k_{s+1}$，或者是一对复共轭的值 $k_s$ 和 $k_{s+1}=k_s^*$。这就给出了一个实矩阵 $\mathbf{A}_{s+2}$

$$\mathbf{A}_{s+2}=\mathbf{Q}_{s+1}\cdot\mathbf{Q}_s\cdot\mathbf{A}_s\cdot\mathbf{Q}_s^T\cdot\mathbf{Q}_{s+1}^T \tag{11.6.4}$$

其中的 $\mathbf{Q}$ 由下列关系确定

$$\mathbf{A}_s-k_s\mathbf{1}=\mathbf{Q}_s^T\cdot\mathbf{R}_s \tag{11.6.5}$$

$$\mathbf{A}_{s+1}=\mathbf{Q}_s\cdot\mathbf{A}_s\cdot\mathbf{Q}_s^T \tag{11.6.6}$$

$$\mathbf{A}_{s+1}-k_{s+1}\mathbf{1}=\mathbf{Q}_{s+1}^T\cdot\mathbf{R}_{s+1} \tag{11.6.7}$$

应用式(11.6.6)，等式(11.6.7)可重写成

$$\mathbf{A}_s-k_{s+1}\mathbf{1}=\mathbf{Q}_s^T\cdot\mathbf{Q}_{s+1}^T\cdot\mathbf{R}_{s+1}\cdot\mathbf{Q}_s \tag{11.6.8}$$

于是，如果我们定义

$$\mathbf{M}=(\mathbf{A}_s-k_{s+1}\mathbf{1})\cdot(\mathbf{A}_s-k_s\mathbf{1}) \tag{11.6.9}$$

则等式(11.6.5)和(11.6.8)给出

$$\mathbf{R} = \mathbf{Q} \cdot \mathbf{M} \tag{11.6.10}$$

其中

$$\mathbf{Q} = \mathbf{Q}_{s+1} \cdot \mathbf{Q}_s \tag{11.6.11}$$

$$\mathbf{R} = \mathbf{R}_{s+1} \cdot \mathbf{R}_s \tag{11.6.12}$$

等式(11.6.4)可重写成

$$\mathbf{A}_s \cdot \mathbf{Q}^T = \mathbf{Q}^T \cdot \mathbf{A}_{s+2} \tag{11.6.13}$$

于是,假设我们能通过某种方法找到一个上海森伯格矩阵 $\mathbf{H}$,满足

$$\mathbf{A}_s \cdot \bar{\mathbf{Q}}^T = \bar{\mathbf{Q}}^T \cdot \mathbf{H} \tag{11.6.14}$$

其中 $\bar{\mathbf{Q}}$ 是正交矩阵。如果 $\bar{\mathbf{Q}}^T$ 与 $\mathbf{Q}^T$ 有相同的第一列(也就是 $\bar{\mathbf{Q}}$ 与 $\mathbf{Q}$ 有相同的第一行),则 $\bar{\mathbf{Q}} = \mathbf{Q}$ 并且 $\mathbf{A}_{s+2} = \mathbf{H}$。

$\mathbf{Q}$ 的第一行可以按如下方法找到。等式(11.6.10)表明 $\mathbf{Q}$ 是将实矩阵 $\mathbf{M}$ 三角化的正交矩阵。任何一个矩阵可以通过左乘一系列豪斯贺德矩阵而三角化。这一系列豪斯贺德矩阵是 $\mathbf{P}_1$(作用在第一列上),$\mathbf{P}_2$(作用在第二列上),…,$\mathbf{P}_{n-1}$。于是 $\mathbf{Q} = \mathbf{P}_{n-1} \cdots \mathbf{P}_2 \cdot \mathbf{P}_1$,而 $\mathbf{Q}$ 的第一行就是 $\mathbf{P}_1$ 的第一行,因为 $\mathbf{P}_i$ 的左上角是 $(i-1)\times(i-1)$ 的单位矩阵。我们现在必须去找满足式(11.6.14),并且与 $\mathbf{P}_1$ 有同样的第一行的 $\bar{\mathbf{Q}}$。

豪斯贺德矩阵 $\mathbf{P}_1$ 是由 $\mathbf{M}$ 的第一列决定的。因为 $\mathbf{A}_s$ 是上海森伯格形式的,等式(11.6.9)表明 M 的第一列有形式 $[p_1, q_1, r_1, 0, \ldots, 0]^T$,其中

$$\begin{aligned} p_1 &= a_{11}^2 - a_{11}(k_s + k_{s+1}) + k_s k_{s+1} + a_{12}a_{21} \\ q_1 &= a_{21}(a_{11} + a_{22} - k_s - k_{s+1}) \\ r_1 &= a_{32}a_{21} \end{aligned} \tag{11.6.15}$$

因此

$$\mathbf{P}_1 = \mathbf{1} - 2\mathbf{w}_1 \cdot \mathbf{w}_1^T \tag{11.6.16}$$

其中 $\mathbf{w}_1$ 只有前三个元素非零(等式11.2.5)。于是矩阵 $\mathbf{P}_1 \cdot \mathbf{A}_s \cdot \mathbf{P}_1^T$ 除去三个多余元素之外是上海森伯格形式:

$$\mathbf{P}_1 \cdot \mathbf{A}_1 \cdot \mathbf{P}_1^T = \begin{bmatrix} \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times \\ \mathbf{x} & \times & \times & \times & \times & \times & \times \\ \mathbf{x} & \mathbf{x} & \times & \times & \times & \times & \times \\ & & & \times & \times & \times & \times \\ & & & & \times & \times & \times \\ & & & & & \times & \times \end{bmatrix} \tag{11.6.17}$$

这个矩阵可以通过一系列豪斯贺德相似变换重新变成上海森伯格形式,而不影响它的第一行。这样的豪斯贺德矩阵中的第一个,$\mathbf{P}_2$,作用在第一列第二、三和四元素上,并将第三和第四元素零化。这就产生了一个和式(11.6.17)形式相同的矩阵,只是三个多余元素向后推了一列:

$$
\begin{bmatrix}
\times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times \\
 & \times & \times & \times & \times & \times & \times \\
 & \mathbf{x} & \times & \times & \times & \times & \times \\
 & \mathbf{x} & \mathbf{x} & \times & \times & \times & \times \\
 & & & & \times & \times & \times \\
 & & & & & \times & \times
\end{bmatrix}
\tag{11.6.18}
$$

这样继续直到 $\mathbf{P}_{n-1}$，我们看到每一步的豪斯贺德矩阵有一个只有第 $r$、$r+1$ 和 $r+2$ 元素为非零的向量 $\mathbf{w}_r$。这些元素是由当前矩阵的第 $(r-1)$ 列的第 $r$、$r-1$ 和 $r+2$ 元素决定的。注意 $\mathbf{P}$ 与 $\mathbf{P}_2,\cdots,\mathbf{P}_{n-1}$ 有相同的结构。

结果是

$$
\mathbf{P}_{n-1}\ldots\mathbf{P}_2\cdot\mathbf{P}_1\cdot\mathbf{A}_s\cdot\mathbf{P}_1^T\cdot\mathbf{P}_2^T\cdot\ldots\mathbf{P}_{n-1}^T = \mathbf{H} \tag{11.6.19}
$$

其中 $\mathbf{H}$ 是上海森柏格形式。于是

$$
\overline{\mathbf{Q}} = \mathbf{Q} = \mathbf{P}_{n-1}\ldots\mathbf{P}_2\cdot\mathbf{P}_1 \tag{11.6.20}
$$

和

$$
\mathbf{A}_{s+2} = \mathbf{H} \tag{11.6.21}
$$

每一步的原点位移都取为当前矩阵 $\mathbf{A}_s$ 的右下角处 $2\times2$ 矩阵的特征值。由此给出

$$
\begin{aligned}
k_s + k_{s+1} &= a_{n-1,n-1} + a_{nn} \\
k_s k_{s+1} &= a_{n-1,n-1}a_{nn} - a_{n-1,n}a_{n,n-1}
\end{aligned}
\tag{11.6.22}
$$

将式(11.6.22)代入式(11.6.15)，我们得到

$$
\begin{aligned}
p_1 &= a_{21}\{[(a_{nn}-a_{11})(a_{n-1,n-1}-a_{11}) - a_{n-1,n}a_{n,n-1}]/a_{21} + a_{12}\} \\
q_1 &= a_{21}[a_{22} - a_{11} - (a_{nn}-a_{11}) - (a_{n-1,n-1}-a_{11})] \\
r_1 &= a_{21}a_{32}
\end{aligned}
\tag{11.6.23}
$$

我们已经明智地归类某些项，以减小当存在小的非对角元时可能带来的舍入误差。由于豪斯贺德矩阵只和元素间的比例有关，我们可以略去式(11.6.23)中的因子 $a_{21}$。

总之，为完成双重步 $QR$ 迭代，我们构造一系列的豪斯贺德矩阵 $\mathbf{P}_r$，$r=1,\ldots,n-1$。对于 $\mathbf{P}_1$ 我们使用式(11.6.23)给出的 $p_1$，$q_1$ 和 $r_1$。对于剩下的矩阵，$p_r$，$q_r$ 和 $r_r$ 由当前矩阵的 $(r,r-1)$、$(r+1,r-1)$ 和 $(r+2,r-1)$ 元素确定。可以通过将豪斯贺德矩阵 $2\mathbf{w}\cdot\mathbf{w}^T$ 部分的非零元素写成如下形式，则减少算术操作的次数

$$
2\mathbf{w}\cdot\mathbf{w}^T = \begin{bmatrix} (p\pm s)/(\pm s) \\ q/(\pm s) \\ r/(\pm s) \end{bmatrix}\cdot\begin{bmatrix} 1 & q/(p\pm s) & r/(p\pm s) \end{bmatrix} \tag{11.6.24}
$$

其中

$$
s^2 = p^2 + q^2 + r^2 \tag{11.6.25}
$$

(我们简单地将每一元素除以一个归一化因子，参见第11.2节中的公式。)

按这种方法进行，收敛通常是很快的。对于一个特征值有两种可能的方式可以结束所进行的迭代。首先，如果 $a_{n,n-1}$ 变成是“可忽略的”，则 $a_{nn}$ 是一个特征值。我们于是可以删去矩阵的第 $n$ 行和第 $n$ 列并继续寻找下一个本征值。第一种情况，$a_{n-1,n-2}$ 可能变得可忽略。这时右

下角的 2×2 矩阵的特征值就是原来矩阵的特征值。我们删去第 $n$ 和第 $n-1$ 行和列，并继续进行下面的计算。

对特征值收敛性的检验与可忽略的次对角元素的检验相结合。这种可忽略的次对角元素使矩阵能分裂成子矩阵。在那些可忽略的 $a_{i,i-1}$ 中寻找 $i$ 值最大的一个，如果 $i=n$，我们就找到了一个特征值；如果 $i=n-1$，我们就找到了两个特征值。否则，我们就继续进行对于子矩阵从第 $i$ 行到第 $n$ 行的迭代(如果没有小的次对角元素，$i$ 就设为单位 1)。

确定 $i$ 值之后，检查子矩阵从第 $i$ 行到第 $n$ 行，是否有两个连续次对角元素的乘积足够小，以至于我们能针对一个更小的子矩阵进行计算。例如，在第 $m$ 行，我们从 $m=n-2$ 开始一直下降到 $i+1$，用式(11.6.23)计算 $p$、$q$ 和 $r$。但要将公式中的 1 用 $m$ 代替，2 用 $m-1$ 代替。如果这些是一个双重步 $QR$ 中特殊的"第一"豪斯贺德矩阵元素，使用这一豪斯贺德矩阵将会导致$(m+1,m-1)$、$(m+2,m-1)$和$(m+2,m)$位置上的元素为非零。我们要求这些元素中的前两个元素与相邻的对角元素 $a_{m-1,m-1}$、$a_{mm}$和 $a_{m+1,m+1}$比较起来较小。一个令人满意的近似标准是

$$|a_{m,m-1}|(|q|+|r|) \ll |p|(|a_{m+1,m+1}|+|a_{mm}|+|a_{m-1,m-1}|) \quad (11.6.26)$$

上述过程很少是不收敛的。经验告诉我们，如果一个双重步 $QR$ 迭代中的位移与矩阵范数是同一量级的，那么通常收敛很快。相应地，如果 10 次迭代还没有确定出一个特征值，下次迭代通常的位移就要换成由下式定义的新的位移

$$\begin{aligned} k_s+k_{s-1} &= 1.5\times(|a_{n,n-1}|+|a_{n-1,n-2}|) \\ k_s k_{s-1} &= (|a_{n,n-1}|+|a_{n-1,n-2}|)^2 \end{aligned} \quad (11.6.27)$$

其中因子 1.5 可以任意挑选，以减少无效的移动。在 20 次不成功迭代后，这一策略重复使用。如果 30 次迭代不成功，则程序宣告失败。

这里描述的 $QR$ 算法的运算量合计为每次迭代 $\sim 5k^2$，其中 $k$ 是当前矩阵的阶数。每个特征值的总运算量 $\sim 3n^3$。这一估计忽略了矩阵分裂或稀疏性可能带来的效率。

下列程序 $hqr$ 是基于上述算法，这来源于文献[1,2]。

```
#include <math.h>
#include "nrutil.h"

void hqr(float **a, int n, float wr[], float wi[])
    寻找上海森伯格矩阵 a[1..n][1..n]的所有特征值。输入的 a 可以是第11.5节中 elmhes 的输出；输出时 a 的原值已
    被破坏。特征值的实部和虚部分别由 wr[1..n]和 wi[1..n]返回。
{
    int nn,m,l,k,j,its,i,mmin;
    float z,y,x,w,v,u,t,s,r,q,p,anorm;

    anorm=fabs(a[1][1]);                  计算矩阵范数，有可能将它用来定位小的单一
    for (i=2;i<=n;i++)                        的次对角元素
        for (j=(i-1);j<=n;j++)
            anorm += fabs(a[i][j]);
    nn=n;
    t=0.0;                                只有一个额外的位移才发生变化
    while (nn >= 1) {                     开始寻找下一个特征值
        its=0;
        do {
            for (l=nn;l>=2;l--) {         开始迭代：寻找小的单一的次对角元素
                s=fabs(a[l-1][l-1])+fabs(a[l][l]);
```

```
        if (s == 0.0) s=anorm;
        if ((float)(fabs(a[l][l-1]) + s) == s) break;
    }
    x=a[nn][nn];
    if (l == nn) {                   找到一个根
        wr[nn]=x+t;
        wi[nn--]=0.0;
    } else {
        y=a[nn-1][nn-1];
        w=a[nn][nn-1]*a[nn-1][nn];
        if (l == (nn-1)) {           找到两个根
            p=0.5*(y-x);
            q=p*p+w;
            z=sqrt(fabs(q));
            x += t;
            if (q >= 0.0) {          ……一对实数根
                z=p+SIGN(z,p);
                wr[nn-1]=wr[nn]=x+z;
                if (z) wr[nn]=x-w/z;
                wi[nn-1]=wi[nn]=0.0;
            } else {                 ……一对复数根
                wr[nn-1]=wr[nn]=x+p;
                wi[nn-1]= -(wi[nn]=z);
            }
            nn -= 2;
        } else {                     没有找到根，继续迭代
            if (its == 30) nrerror("Too many iterations in hqr");
            if (its == 10 || its == 20) {        形成特殊的位移
                t += x;
                for (i=1;i<=nn;i++) a[i][i] -= x;
                s=fabs(a[nn][nn-1])+fabs(a[nn-1][nn-2]);
                y=x=0.75*s;
                w = -0.4375*s*s;
            }
            ++its;
            for (m=(nn-2);m>=l;m--) {            形成位移并寻找2个连续的小
                z=a[m][m];                          的次对角元素
                r=x-z;
                s=y-z;
                p=(r*s-w)/a[m+1][m]+a[m][m+1];   (11.6.23)式
                q=a[m+1][m+1]-z-r-s;
                r=a[m+2][m+1];
                s=fabs(p)+fabs(q)+fabs(r);       为防止上溢出或下溢出而改变
                p /= s;                             尺度
                q /= s;
                r /= s;
                if (m == l) break;
                u=fabs(a[m][m-1])*(fabs(q)+fabs(r));
                v=fabs(p)*(fabs(a[m-1][m-1])+fabs(z)+fabs(a[m+1][m+1]));
                if ((float)(u+v) == v) break;    (11.6.26)式
            }
            for (i=m+2;i<=nn;i++) {
                a[i][i-2]=0.0;
                if (i != (m+2)) a[i][i-3]=0.0;
            }
            for (k=m;k<=nn-1;k++) {
            对从第l行到第nn行及从第m列到第nn列施行双QR步骤
                if (k != m) {
                    p=a[k][k-1];                 开始设置豪斯荷德向量
                    q=a[k+1][k-1];
                    r=0.0;
                    if (k != (nn-1)) r=a[k+2][k-1];
                    if ((x=fabs(p)+fabs(q)+fabs(r)) != 0.0) {
                        p /= x;                  为防止上溢出或下溢出而改变
                        q /= x;                     尺度
                        r /= x;
                    }
```

```
                    }
                    if ((s=SIGN(sqrt(p*p+q*q+r*r),p)) != 0.0) {
                        if (k == m) {
                            if (l != m)
                            a[k][k-1] = -a[k][k-1];
                        } else
                            a[k][k-1] = -s*x;
                        p += s;                           (11.6.24)式
                        x=p/s;
                        y=q/s;
                        z=r/s;
                        q /= p;
                        r /= p;
                        for (j=k;j<=nn;j++) {             行调整
                            p=a[k][j]+q*a[k+1][j];
                            if (k != (nn-1)) {
                                p += r*a[k+2][j];
                                a[k+2][j] -= p*z;
                            }
                            a[k+1][j] -= p*y;
                            a[k][j] -= p*x;
                        }
                        mmin = nn<k+3 ? nn : k+3;
                        for (i=l;i<=mmin;i++) {           列调整
                            p=x*a[i][k]+y*a[i][k+1];
                            if (k != (nn-1)) {
                                p += z*a[i][k+2];
                                a[i][k+2] -= p*r;
                            }
                            a[i][k+1] -= p*q;
                            a[i][k] -= p;
                        }
                    }
                }
            }
        }
    } while (l < nn-1);
  }
}
```

参考文献和进一步读物：


Wilkinson, J. H, and Reinsch, C, 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [1]

Smith, B. T, et al. 1976, *Matrix Eigensystem Routines-EISPACK Guide*, 2nd ed, vol. 6 of Lecture Notes in Computer Science (New York: Spring-Verlag). [2]


## 11.7 用逆迭代法改进特征值并寻找特征向量

逆迭代的基本思想很简单。令 $\mathbf{y}$ 是线性系统的解

$$(\mathbf{A} - \tau \mathbf{1}) \cdot \mathbf{y} = \mathbf{b} \tag{11.7.1}$$

其中 $\mathbf{b}$ 是一个随机向量，而 $\tau$ 靠近 $\mathbf{A}$ 的某一特征值 $\lambda$。这一过程可迭代进行；用 $\mathbf{y}$ 代替 $\mathbf{b}$ 并解出新的 $\mathbf{y}$，新的 $\mathbf{y}$ 更接近真实的特征向量。

我们可以通过将 $\mathbf{y}$ 和 $\mathbf{b}$ 展开成 $\mathbf{A}$ 的特征向量 $\mathbf{x}_j$ 的线性组合，来看看逆迭代法的工作原理。

$$\mathbf{y} = \sum_j \alpha_j \mathbf{x}_j$$
$$\mathbf{b} = \sum_j \beta_j \mathbf{x}_j \tag{11.7.2}$$

于是式(11.7.1)给出

$$\sum_j \alpha_j(\lambda_j - \tau)\mathbf{x}_j = \sum_j \beta_j \mathbf{x}_j \tag{11.7.3}$$

从而有

$$\alpha_j = \frac{\beta_j}{\lambda_j - \tau} \tag{11.7.4}$$

和

$$\mathbf{y} = \sum_j \frac{\beta_j \mathbf{x}_j}{\lambda_j - \tau} \tag{11.7.5}$$

如果 $\tau$ 接近 $\lambda_n$，则只要 $\beta_n$ 不是意外地太小，$\mathbf{y}$ 将近似于某种归一化的 $\mathbf{x}_n$。进而，这一过程的每一步迭代都给出式(11.7.5)中的分母 $(\lambda_j-\tau)$ 的一个幂次。所以，对于分得很开的特征值其收敛很快。

假设在第 $k$ 步迭代，我们解等式

$$(\mathbf{A} - \tau_k \mathbf{1}) \cdot \mathbf{y} = \mathbf{b}_k \tag{11.7.6}$$

其中 $\mathbf{b}_k$ 和 $\tau_k$ 是我们感兴趣的特征向量和特征值的当前估计(例如，$\mathbf{x}_n$ 和 $\lambda_n$)。归一化 $\mathbf{b}_k$，使 $\mathbf{b}_k \cdot \mathbf{b}_k = 1$。精确的特征向量和特征值满足

$$\mathbf{A} \cdot \mathbf{x}_n = \lambda_n \cdot \mathbf{x}_n \tag{11.7.7}$$

于是

$$(\mathbf{A} - \tau_k \mathbf{1}) \cdot \mathbf{x}_n = (\lambda_n - \tau_k)\mathbf{x}_n \tag{11.7.8}$$

因为式(11.7.6)中的 $\mathbf{y}$ 是 $\mathbf{x}_n$ 的进一步近似，我们将其归一化并令

$$b_{k+1} = \frac{\mathbf{y}}{|\mathbf{y}|} \tag{11.7.9}$$

通过将改进的猜测值 $\mathbf{y}$ 代入式(11.7.8)，得到特征值的一个改进的估计值。根据式(11.7.6)，左边是 $\mathbf{b}_k$，将 $\lambda_n$ 称作新值 $\tau_{k+1}$，我们发现

$$\tau_{k+1} = \tau_k + \frac{1}{\mathbf{b}_k \cdot \mathbf{y}} \tag{11.7.10}$$

上述公式看起来很简单，但实际计算中可能很棘手。首先，一个需要重新考虑的问题就是何时使用逆迭代。大多数工作量是解线性系统(11.7.6)。所以一个可能的策略是，首先将矩阵 $\mathbf{A}$ 约化成一个简单的形式，使得式(11.7.6)更容易求解。显然，这样的形式对于对称矩阵来说是三对角形式，而对于非称矩阵来说是海森伯格形式。然后，使用逆迭代来产生所有的特征向量。这种方法对于对称矩阵的运算量为 $O(N^3)$，它远不及前面提到的 $QL$ 方法有效。实际上，一旦要求计算四分之一以上数量的特征向量，即使是最好的逆迭代程序也不及 $QL$ 方法有效。所以只在当已知有较好的特征值，并只需求其中几个选出的特征向量时，才使用逆迭代方法。

读者可以自己写一个简单的逆代迭代程序，使用 $LU$ 分解来解式(11.7.6)。读者可以自己决定是使用我们在第二章中给出的一般 LU 算法，还是利用三对角或海森伯格矩阵的特殊形式。注意，因为式(11.7.6)的线性系统是近乎奇异的，我们必须小心使用第2.3节中的 $LU$ 分解，它用一个非常小的数来代替零主元。

在本书中，我们不给出一个一般的逆迭代程序。因为要考虑所有可能出现的情况是有些麻烦的。在文献[1,2]中给出了程序。如果你使用这些程序或者自己写程序，下列的指点是很

有用的。

人们在开始的时候，给出一个代表特征值 $\lambda_n$ 的数值，选择一个归一化的随机向量 $\mathbf{b}_0$ 作为对特征向量 $\mathbf{x}_0$ 的初始猜测，并求解式(11.7.6)。新的向量 $\mathbf{y}$ 比 $\mathbf{b}_0$ 大一个"增长因子" $|\mathbf{y}|$，它理所当然应该大些。同样特征值的变化，根据式(11.7.10)本质上应是 $1/|\mathbf{y}|$，它应该小一些。下列情况可能出现：

- 如果开始时增长因子太小，则可以肯定我们选择了一个"坏"的随机向量。这不仅可能是由于式(11.7.5)中的 $\beta_n$ 很小，同样，也可能由于是缺损矩阵情形，这时式(11.7.5)根本不能使用(详细情况参见[1]或[3])。我们应返回到起始并选择一个新的初始向量。
- 变化量 $|\mathbf{b}_1-\mathbf{b}_0|$ 应小于某容差限 $\varepsilon$。我们能用它作为停止迭代的标准，一直迭代到满足此标准，最多要进行 5～10 次迭代。
- 经过几次迭代后，如果 $|\mathbf{b}_{k-1}-\mathbf{b}_k|$ 减小得不够快，我们应根据式(11.7.10)或试着修正特征值。如果在机器精度下 $\tau_{k+1}=\tau_k$，我们就不要再去修正特征向量而应放弃迭代，否则就用新的特征值开始另一轮迭代。

我们不是每一步都修正特征值，其原因是因为用 $LU$ 分解线性系统(11.7.6)时，如果 $\tau_k$ 是固定的则可以节省分解。我们在每次更新 $\mathbf{b}_k$ 时，只需做回代。对于具有固定 $\tau_k$ 的迭代次数是以下两者的折衷，或者以二次收敛但由于每次改进 $\tau_k$ 带来 $O(N^3)$ 的运算量，或者以线性收敛但由于保持固定的 $\tau_k$ 所需运算量为 $O(N^2)$。如果已经应用本章前面给出的某个程序确定了特征值，则它可能修正到机器的精度，于是就不必更新它。

有两种病态情况可能会在逆迭代过程中出现。一种是重根或离得很近的根。这是对称阵情况中为常见的问题。逆迭代对于一个给定的初始猜测 $\tau_0$ 只能找到一个特征向量。一个好的策略是，扰动 $\tau_0$ 的最后几个有效位并再次迭代。这样，常常能得到一个线性无关的特征向量。一般说来，必须采取特殊的步骤来保证线性无关特征向量的正交性。鉴于雅可比和 $QL$ 算法，既使在多重特征值的情况下，也会自动产生正交的特征向量。

第二个问题，特别对于非对称矩阵来说，是关于缺损的问题。除非做出一个"好"的初始估计，不然增长因子会很小。更有甚者，迭代对问题没有改进。在这种情况下，解决的办法是选择随机的初始向量，解一次式(11.7.6)，并只要任何向量给出一个可接受的大小的增长因子，则就停止迭代。通常只需进行几次试验。

对于非对称矩阵的一个进一步的复杂情况是实矩阵可能有一对复共轭的特征值。这时将不得不使用复数算法从式(11.7.6)解出复向量。对于任何中型(或大型)非对称矩阵，建议避免使用逆迭代法，而选用包括复算法的 $QR$ 方法。读者可以从文献[1,2]和其它地方找到这类程序。

参考文献和进一步读物：

Wilkinson, J. H. and Reinxsch, C. 1971. *Linear Algebra*, vol. Ⅱ of *Handbook for Automatic Computation* (New York: Springer-Verlag), p. 418. [1]

Smith, B. T., et al. 1976, *Matrix Eigensystem Routines-EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag). [2]

# 第十二章　快速傅里叶变换

## 12.0　引言

有一大类重要的计算问题归入一般的"傅里叶(Fourier)变换法"或"谱分析法"范畴之内。对于其中某些计算问题来说,傅里叶变换只是一种计算工具,是完成数据的某种普通操作的有效计算工具。而对于其它一些情况,人们关心的问题却是傅里叶变换(或有关的"功率谱")其自身内在的含意是什么。这两种问题具有一种共同的方法论。

主要由于历史原因,使傅里叶方法和谱分析法的文献一直与"古典的"数值分析的文献没有什么联系。但在当今的时代,就毫无理由再这样割离开了。傅里叶方法在研究中是极为平凡的方法,我们也不再视其为高深莫测或秘不可宣的了。可同时,我们也感觉到许多计算机用户在这领域内的经验比其它领域,例如微分方程或数值积分领域的经验相对地要少些。所以,我们对解析结果的分析要做得更完整些。数值算法基本上将在第12.2节开始,而傅里叶变换方法的各种应用将放在十三章中讨论。

一个物理过程既可以在**时域**内,通过把物理量 $h$ 作为时间 $t$ 的函数 $h(t)$ 来描述;也可以在**频域**内,通过将振幅 $H$(通常是一个包含相位的复数)作为频率 $f$ 的函数 $H(f)$ 来描述,其中 $-\infty<f<\infty$。在许多情况下,把 $h(t)$ 和 $H(f)$ 考虑为同一函数的两种不同的表达方式是很有用的。根据**傅里叶变换**方程便可实现这两个表达之间的相互转变。

$$H(f)=\int_{-\infty}^{\infty}h(t)e^{2\pi ift}dt$$
$$h(t)=\int_{-\infty}^{\infty}H(f)e^{-2\pi ift}df \tag{12.0.1}$$

如果 $t$ 以秒为单位,则方程式(12.0.1)中 f 是以周期每秒或赫兹(Hertz)为单位(赫兹是频率单位)。当然,方程也可采用其它单位。如果 $h$ 是位移 $x$(米)的函数,$H$ 则为反波长(周期数/每米)的函数,等等。若读者是一位训练有素的物理学家或数学家,就可能更习惯于使用角频率 $\omega$(单位为弧度/秒)。$\omega$ 和 $f$、$H(\omega)$和 $H(f)$之间的关系为

$$\omega\equiv 2\pi f\qquad H(\omega)\equiv[H(f)]_{f=\omega/2\pi} \tag{12.0.2}$$

则方程式(12.0.1)可写成如下形式

$$H(\omega)=\int_{-\infty}^{\infty}h(t)e^{i\omega t}dt$$
$$h(t)=\frac{1}{2\pi}\int_{-\infty}^{\infty}H(\omega)e^{-i\omega t}d\omega \tag{12.0.3}$$

我们开始时使用的是这习惯单位 $\omega$,可后来我们有所改变!因为若使用 $f$ 单位,则有关 $2\pi$ 的因式便可不用记住了,特别是在第12.1节中当我们处理离散样本数据时更是如此。

从方程式(12.0.1)中,可明显看出傅里叶变换是线性运算。两个函数之和的变换应等于函数变换之和。一个常数乘以一个函数之积的变换,应等于同一常数乘以函数变换的积。

在时域中，函数 $h(t)$ 可能具有一个或多个特殊对称性。它可能是**纯实函数**或者**纯虚函数**，也可能是偶函数 $h(t)=h(-t)$，或者**奇函数** $h(t)=-h(t)$。在频域中，这些对称性导出了 $H(f)$ 和 $H(-f)$ 之间的关系。下表列出了在时域和频域中对称性的对应关系：

| 假如… | 则 |
|---|---|
| $h(t)$ 是实函数 | $H(-f)=[H(f)]^*$ |
| $h(t)$ 是虚函数 | $H(-f)=-[H(f)]^*$ |
| $h(t)$ 是偶函数 | $H(-f)=H(f)$（即 $H(f)$ 是偶函数） |
| $h(t)$ 是奇函数 | $H(-f)=-H(f)$（即 $H(f)$ 是奇函数） |
| $h(t)$ 是实偶函数 | $H(f)$ 是实偶函数 |
| $h(t)$ 是实奇函数 | $H(f)$ 是虚奇函数 |
| $h(t)$ 是虚偶函数 | $H(f)$ 是虚偶函数 |
| $h(t)$ 是虚奇函数 | $H(f)$ 是实奇函数 |

在以后的章节中，我们将会看到如何利用这些对称性来提高计算效率。

下面还有傅里叶变换的一些其它的基本性质。（我们用符号"$\Leftrightarrow$"表示一个变换对。）如果

$$h(t)\Leftrightarrow H(f)$$

是一对变换对，则其它变换对为

$$h(at)\Leftrightarrow\frac{1}{|a|}H\left(\frac{f}{a}\right)\qquad\text{“时间标度变换”}\tag{12.0.4}$$

$$\frac{1}{|b|}h\left(\frac{t}{b}\right)\Leftrightarrow H(bf)\qquad\text{“频率标度变换”}\tag{12.0.5}$$

$$h(t-t_0)\Leftrightarrow H(f)e^{2\pi i f t_0}\qquad\text{“时间平移”}\tag{12.0.6}$$

$$h(t)e^{-2\pi i f_0 t}\Leftrightarrow H(f-f_0)\qquad\text{“频率平移”}\tag{12.0.7}$$

利用两个函数 $h(t)$、$g(t)$ 和它们相应的傅里叶变换 $H(f)$、$G(f)$，我们可以组成两种情况别有意义的组合。这两个函数的**卷积**记作 $g*h$，其定义为：

$$g*h\equiv\int_{-\infty}^{\infty}g(\tau)h(t-\tau)d\tau\tag{12.0.8}$$

注意，$g*h$ 是一个时域函数，而且 $g*h=h*g$。由此推导出函数 $g*h$ 是以下简单变换对中的一员。

$$g*h\Leftrightarrow G(f)H(f)\qquad\text{“卷积定理”}\tag{12.0.9}$$

换言之，卷积的傅里叶变换恰好等于傅里叶变换的乘积。

这两个函数的**相关函数**，记作 $\mathrm{Corr}(g,h)$，其定义为

$$\mathrm{Corr}(g,h)\equiv\int_{-\infty}^{\infty}g(\tau+t)h(\tau)d\tau\tag{12.0.10}$$

该相关函数是 $t$ 的函数，称 $t$ 为滞后时间。所以，它依赖于时域，而且可以证明它是以下变换对中的一员

$$\mathrm{Corr}(g,h)\Leftrightarrow G(f)H^*(f)\qquad\text{“相关定理”}\tag{12.0.11}$$

[更一般而言，这变换对中的另一员是 $G(f)H(-f)$，但通常情况下我们限制 $g$ 与 $h$ 都为实函数，所以我们冒昧地特设 $H(-f)=H^*(f)$]。这个结果证明，一个函数的傅氏变换乘以另一函数傅氏变换的复共轭就得到它们的相关函数的傅氏变换。一个函数与其自身的相关称

为它的**自相关**。在这情况下，式(12.0.11)成为以下变换对

$$\mathrm{Corr}(g,g) \Leftrightarrow |G(f)|^2 \qquad \text{"Wiener-Khinchin 定理"} \tag{12.0.12}$$

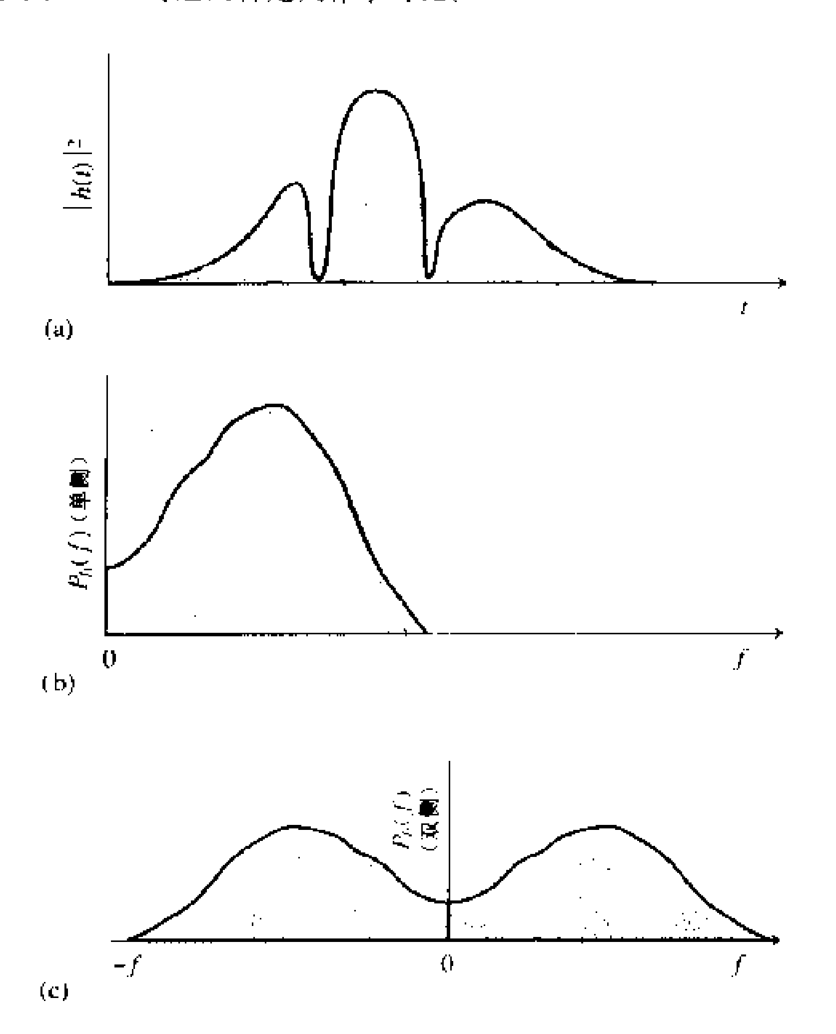
无论我们是在时域还是在频域里计算，一个信号的**总功率**总是相同的。这一结论就是**帕斯维尔**(Parserval)定理：

$$\text{总功率} \equiv \int_{-\infty}^{\infty} |h(t)|^2 dt = \int_{-\infty}^{\infty} |H(f)|^2 \, df \tag{12.0.13}$$

通常，人们想知道包含在 $f$ 和 $f-df$ 频率区间内的"功率大小"。在这种情况下，人们通常不区分正、负频率 $f$，而宁愿认为 $f$ 是从0("零频率"或直流)变化到 $+\infty$。这时，人们定义函数 $h$ 的**单侧功率谱密度**(PSD)为

$$P_h(f) \equiv |H(f)|^2 + |H(-f)|^2 \qquad 0 \leq f < \infty \tag{12.0.14}$$

因此，总功率巧好是 $P_h(f)$ 从 $f=0$ 到 $f=\infty$ 的积分。若函数 $h(t)$ 为实函数，则式(12.0.14)中两项相等，故 $P_h(f)=2|H(f)|^2$。要告诫的是，人们会偶尔看到所定义的 PSD 没有这个因子 2 。严格地说，这应称为**双侧功率谱密度**，可是有些书并未明确地指出所假定的是单侧还是双侧谱密度，图12.0.1将这两种定义作了对比。




函数平方的曲线下的面积，如(a)；它等于在正频率域上单侧功率谱密度曲线下的面积，如(b)；也等于正、负频率域上双侧功率密度曲线下的面积，如(c)。

图12.0.1 单侧和双侧功率谱的规范图

如果函数 $h(t)$ 无限地在 $-\infty<t<\infty$ 中变化，则一般而言总功率和功率谱密度将是无穷的。于是人们感兴趣的就是**单位时间的(单侧或双侧)功率谱密度**。通过选取函数 $h(t)$ 的一个长而有限的截段来计算它的 PSD(即，计算在有限长度截段内等于 $h(t)$，而其它区域等于零的一个函数的 PSD)，然后，用所取的截段长度除以所得的 PSD。在这种情况下，帕斯尔定理表明，在正频率域上单位时间的单侧 PSD 的积分值等于信号 $h(t)$ 的均方振幅值。

读者可能会担心，单位时间的 PSD 是频率 f 的函数，那么当人们用越来越长的数据截段来求它的值时，它的收敛性如何？这个有趣的问题正是"功率谱估计"主题中的内容，将在后面第13.4节～第13.7节中讨论。现在只给出一个粗略回答：在所有频率上，除了那些 $h(t)$ 具有有限振幅的离散正弦波(或余弦波)分量的频率以外，单位时间的 PSD 收敛到有限值。在其它频率上，它成为一个 $\delta$ 函数，即一个陡直尖峰，它的宽度越来越窄，但它的面积却收敛到该频率离散正弦分量的均方振幅值。

迄今为止，我们已阐述了本章将用到的所有解析公式，只有一个例外：即在计算工作中，特别是在带实验数据的计算工作中，我们几乎从来不给出一个连续函数 $h(t)$ 来计算，而是给出离散集 $t_i$ 上的一系列测量值 $h(t_i)$。在这看来似乎不重要的事实中，所包含的深奥内容正是下一节讨论的主题。

## 12.1 离散样本数据的傅里叶变换

在最普通情况下，函数 $h(t)$ 在等距时间间隔上被取样(即在这些时间间隔上函数值被记录)。设 $\Delta$ 表示相邻样本间的时间间隔，于是样本值的序列为：

$$h_n = h(n\Delta) \qquad n = \ldots, -3, \quad -2, \quad -1, 0, 1, 2, 3, \ldots \qquad (12.1.1)$$

时间间隔 $\Delta$ 的倒数称为**取样率**。例如，如果 $\Delta$ 用秒度量，则取样率就是每秒记录的样本的个数。

### 12.1.1 取样定理与混叠现象

对于任何取样间隔 $\Delta$，都有一个特殊的频率 $f_c$，称之为**奈奎斯特**(Nyquist)**临界频率**，给出如下

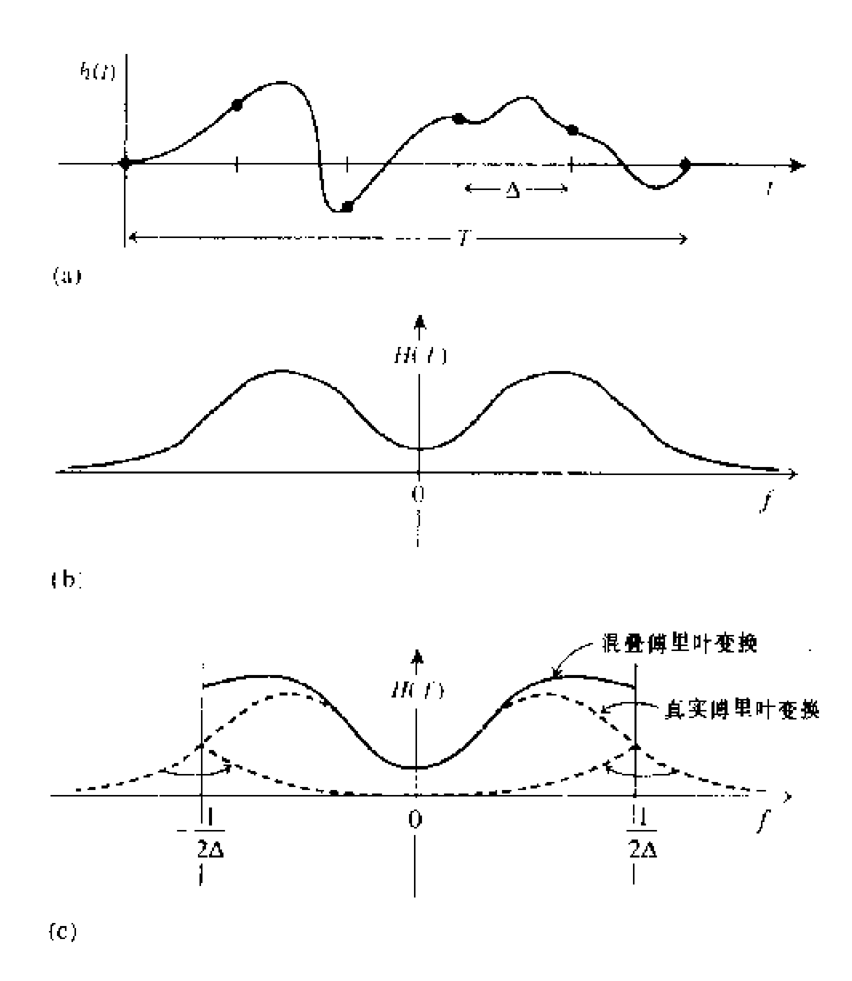
$$f_c = \frac{1}{2\Delta} \qquad (12.1.2)$$

如果具有奈奎斯特临界频率的一个正弦波在它的正波峰值处被取样，则下一个样本值将取在负波谷处，然后样本值再一次取在正波峰值上，以此类推。也可以说是：**一个正弦波的临界取样是每个周期有两个样本点**。人们常常选用取样间隔 $\Delta$ 的单位来测量时间，在这种情况下，奈奎斯特临界频率恰好是常数 1/2 。

奈奎斯特临界频率之所以重要，是因为有两个彼此相关又互不相同的理由。一个理由是件有益的事，另一个理由是件不利的事。首先谈谈这件有益的事，这就是众所周知的**取样定理**：如果一个连续函数 $h(t)$ 以间隔 $\Delta$ 取样，并且函数的振幅谱**带宽受限**，最高频率小于 $f_c$，即如果对于所有 $|f|<f_c$，$H(f)=0$，则函数 $h(t)$ 可由它的样本值 $h_n$ 完全确定。实际上 $h(t)$ 可用以下公式表示

$$h(t)=\Delta\sum_{n=-\infty}^{+\infty}h_n\frac{\sin[2\pi f_c(t-n\Delta)]}{\pi(t-n\Delta)}\tag{12.1.3}$$

有很多理由可说明这是一个引人注目的定理，理由之一是定理表明，在某种意义上说，一个带宽有限的函数，其"信息量"比普通连续函数的"信息量"要无限地小。通常，人们所处理的这些具有物理背景的信号其宽总是有限的(或至少近似有限带宽)。例如，信号通过一个具有已知有限频率响应的放大器。在这种情况下，取样定理告诉我们，此信号的全部信息量可以通过取样被记录下来，其取样率 $\Delta^{-1}$ 等于所通过的放大器的最高频率的两倍(参看式(12.1.2)。)




(a)中所示的连续函数只在有限时间间隔 $T$ 内不为零，(b)中所示的是图(a)的傅里叶变换，其不是带宽受限的，而是对所有频率都有有限振幅。若原来的函数以取样间隔 Δ 进行取样如(a)中所示，则其样本序列的傅里叶变换如(c)所示，它只定义在正负奈奎斯特临界频率之间，范围以外的功率被折回并混叠到该范围上。只有在取样以前，将原来的函数通过低通滤波后，这种效应才能消除。

图12.1.1　克服混叠效应的方法

接下来谈谈这个不利的理由。它关系到对一个带宽频率不是限制为小于奈奎斯特临界频率的连续函数进行取样的效果。在这种情况下，将出现位于频率范围 $-f_c<f<f_c$ 以外的所有功率谱密度都错误地移入了该频率范围。这一现象称作混叠效应。正是通过离散取样的作用，频率范围$(-f_c,f_c)$以外的任何频率分量都被混淆(错误地转移)进了该区域。读者自

己可以很容易地确信，当且仅当 $f_1$ 和 $f_2$ 间距差是 $1/\Delta$ 的倍数，即这倍数恰好为 $(-f_c, f_c)$ 的频带宽度，则两个波 $\exp(2\pi i f_1 t)$ 和 $\exp(2\pi i f_2 t)$ 在间隔 $\Delta$ 上会给出同样的样本值。这时，一旦对信号进行了离散取样，便无法再消去所混叠的功率。克服混叠效应的方法有：(i)知道信号的自然限制带宽——或者，也可将连续信号通过模拟滤波，强制实行已知带宽的限制。(ii)以足够高的取样率进行取样，以使在最高精度处，每个周期能至少得到两个取样点。上页图 12.1.1 阐明了这些思想。

为了从取样定理角度来观察混叠效应，我们可以持这样的观点：如果一个连续函数已被完全取样，则当我们从离散样本序列来估算它的傅里叶变换时，我们可以假设（或者还是这样假设好）在 $-f_c$ 至 $f_c$ 频率范围之外，其傅里叶变换等于零。于是，我们可根据这一假设观察某函数的傅里叶变换，从而得知这个连续函数是否被完全取样（即混叠效应达到最小）。要做到这一点，我们注意观察，当频率从低趋向 $f_c$ 或者频率从高趋向 $-f_c$ 时，傅里叶变换是否已经趋向于零。相反，若傅里叶变换趋于某个有限值，则说明频率范围以外的分量可能已经混叠到临界频率范围之内了。

### 12.1.2 离散傅里叶变换

现在，我们从有限个函数的样本点来估算这个函数的傅里叶变换。假如我们有 $N$ 个连续相邻的样本值

$$h_k \equiv h(t_k), \quad t_k \equiv k\Delta, \quad k = 0,1,2,\ldots,N-1 \tag{12.1.4}$$

其中 $\Delta$ 为取样间隔。为了使问题简化，我们还假设 $N$ 是偶数。如果函数 $h(t)$ 只在有限时间间隔内不为零，那么应假设整个这有限时间间隔应包容在给出的这 $N$ 个点的区间内。另外，如果函数 $h(t)$ 永远延伸下去，那么应假设这 $N$ 个样本点至少应是 $h(t)$ 的一些典型的样本点，它与其它时间的 $h(t)$ 形式相似。

有了 $N$ 个输入数，我们显然可以得出仅仅是 $N$ 个独立的输出数。因此，我们不必去估算在 $-f_c$ 到 $f_c$ 范围内所有 $f$ 值的傅里叶变换 $H(f)$，而只需在离散值

$$f_n \equiv \frac{n}{N\Delta}, \quad n = -\frac{N}{2},\ldots,\frac{N}{2} \tag{12.1.5}$$

处求 $H(f)$ 估计值。式(12.1.5)中 $n$ 的两个端点值正好对应于奈奎斯特临界频率范围的下限和上限。如果读者很内行，就会注意到式(12.1.5)中，$n$ 的值有 $N+1$ 个，而不是 $N$ 个；由此得出，$n$ 的两个端点值不是无关的（实际上它们是相等的），而其他所有的值是无关的，这便将总数 $n$ 减少到 $N$ 个。

余下的步骤是，有一个离散和式来近似式(12.0.1)中的积分：

$$H(f_n) = \int_{-\infty}^{\infty} h(t)e^{2\pi i f_n t}dt \approx \sum_{k=0}^{N-1} h_k e^{2\pi i f_n t_k}\Delta = \Delta\sum_{k=0}^{N-1} h_k e^{2\pi i kn/N} \tag{12.1.6}$$

此处，等式(12.1.4)和(12.1.5)已用于最末的等式中。我们称式(12.1.6)的最末等式为 $N$ 个点 $h_k$ 的**离散傅里叶变换**，记作为 $H_n$

$$H_n = \sum_{k=0}^{N-1} h_k e^{2\pi i kn/N} \tag{12.1.7}$$

离散傅里叶变换将 $N$ 个复数($h_k$)映成了 $N$ 个复数($H_n$)。它不依赖于任何维参数，例如时标 $\Delta$。当将一组数看成是以间隔 $\Delta$ 取样的连续函数的样本值时，式(12.1.6)给出了这组数的

离散傅里叶变换和函数的连续傅里叶变换之间的关系，这关系式(12.1.6)可以重新写成

$$H(f_n) \approx \Delta H_n \tag{12.1.8}$$

其中 $f_n$ 由式(12.1.5)给出。

迄今为止，我们一直认为式(12.1.7)中的指数 $n$ 由 $-N/2$ 变化到 $N/2$(参看式(12.1.5))。然而，不难看出，在式(12.1.7)中，$n$ 是以周期 $N$ 循环。因此，$H_{-n}=H_{N-n}$ ($n=1,2,\ldots$)。当人们记住了这种变换，一般可设 $H_n$ 中的 $n$ 是从0到 $N-1$ 间变化(一个完全周期)。$n$ 和 $k$($h_k$ 中的)也在同一范围内变化，所以 $N$ 个数到 $N$ 个数的映射是很明显的。若遵循这个约定，就必须记住零频率对应于 $n=0$，正频率 $0<f<f_c$ 对应于值 $1\leqslant n\leqslant N/2-1$，负频率 $-f_c<f<0$ 对应于 $N/2+1\leqslant n\leqslant N-1$。值 $n=N/2$ 既对应于 $f=f_c$，也对应于 $f=-f_c$。

类似于连续傅里叶变换，离散傅里叶变换也具有对称性。例如，在式(12.0.3)之后的表中，只要将 $h(t)$ 改成 $h_k$，$H(f)$ 改成 $H_n$，则表中所有的对称性都成立(同样，时间的"偶"、"奇"性涉及到 $h_k$ 在 $k$ 和 $N-k$ 处的值究竟是恒等还是互为相反数)。

离散傅里叶逆变换就是由 $H_n$ 准确地恢复成这组数组 $h_k$，其公式是

$$h_k = \frac{1}{N}\sum_{n=0}^{N-1} H_n e^{-2\pi i k n/N} \tag{12.1.9}$$

注意(12.1.9)和(12.1.7)两式之间仅有的差别是：(i)改变了指数的符号；(ii)将结果除以 $N$。这意味着，计算离散傅里叶变换的程序在稍加修改后同样也适用于计算逆变换。

帕斯维尔(Parseval)定理的离散形式是

$$\sum_{k=0}^{N-1}|h_k|^2 = \frac{1}{N}\sum_{n=0}^{N-1}|H_n|^2 \tag{12.1.10}$$

卷积定理和相关定理(方程(12.0.9)和(12.0.11))也有离散形式，将在第13.1节和13.2节中分别进行讨论。

## 12.2 快速傅里叶变换(FFT)

计算 $N$ 个点的傅里叶变换，如式(12.1.7)，需要多少计算量？许多年来，直到60年代中期，标准答案是：定义 $W$ 为复数

$$W \equiv e^{2\pi i/N} \tag{12.2.1}$$

则式(12.1.7)可写成

$$H_n = \sum_{k=0}^{N-1} W^{nk} h_k \tag{12.2.2}$$

换言之，向量 $h_k$ 乘以一个矩阵，该矩阵在$(n,k)$处的元素为常数 $W$ 的 $n\times k$ 次幂。这样，矩阵相乘的结果产生一个向量，其分量就是 $H_n$。这个矩阵乘法显然需要 $N^2$ 个复数乘法运算和加上生成复数 $W$ 所需的少量运算。因此，离散傅里叶变换呈现为一个 $O(N^2)$ 过程。可是，这种表面现象是虚假的！事实上，离散傅里叶变换可以用一种称为**快速傅里叶变换**的算法，即 FFT 来计算，只需 $O(N\log_2N)$ 次运算。$N\log_2N$ 和 $N^2$ 间的差别是巨大的。例如，当 $N=10^6$，在一个每秒运算百万次的计算机上，粗略地说，它们之间就是占用 30 秒 CPU 时间和两星期 CPU 时间的差别。直到 60 年代中期，从库利(J. W. Cooley)和图凯(J. W. Tukey)的工作中，人们才逐渐熟悉 FFT 算法。回顾历史，现在我们知道，从 1850 年高斯(Gauss)起，已有

十几个人分别独立地发现了计算 DFT 的有效算法，并在某些情况下得以实现（参考[1]）。

FFT 的"再发现"者是 1942 年丹尼尔森(Danielson)和兰佐斯(Lanczos)俩人，他们提供了算法的一种最清晰的推导。丹尼尔森和兰佐斯证明了一个界长为 $N$ 的离散傅里叶变换可以重新写成两个界长各为 $N/2$ 的离散傅里叶变换之和。其中一个变换由原来 $N$ 个点中的偶数点构成，另一个变换由奇数点构成。证明简单如下：

$$
\begin{aligned}
F_k &= \sum_{j=0}^{N-1} e^{2\pi ijk/N} f_j \\
&= \sum_{j=0}^{N/2-1} e^{2\pi ik(2j)/N} f_{2j} + \sum_{j=0}^{N/2-1} e^{2\pi ik(2j+1)/N} f_{2j+1} \\
&= \sum_{j=0}^{N/2-1} e^{2\pi ikj/(N/2)} f_{2j} + W^k \sum_{j=0}^{N/2-1} e^{2\pi ikj/(N/2)} f_{2j+1} \\
&= F_k^e + W^k F_k^o
\end{aligned}
\tag{12.2.3}
$$

最后一行中，$W$ 与式(12.2.1)中的 $W$ 是同一复常量。$F_k^e$ 是由原 $f_j$ 的偶分量得到的、界长为 $N/2$ 的傅里叶变换的第 $k$ 个分量；而 $F_k^o$ 是由奇分量得到的界长为 $N/2$ 的对应变换。还需注意，式(12.2.3)中最后一行的 k 是从 0 到 $N$ 间变化，而不是仅变到 $N/2$。然而，变换 $F_k^e$ 和 $F_k^o$ 对 $k$ 而言其循环周期为 $N/2$，所以 $F_k^e$ 和 $F_k^o$ 要重复两次循环才能获得所有 $F_k$。

**丹尼尔森-兰佐斯引理**的巧妙之处就是在于它可以递归地使用。在将计算 $F_k$ 的问题简化为计算 $F_k^e$ 和 $F_k^o$ 以后，我们可以再做同样的简化，将 $F_k^e$ 简化成计算它的 $N/4$ 偶数项输入数据的变换和 $N/4$ 奇数项数据的变换。换言之，我们可以将 $F_k^{ee}$ 和 $F_k^{eo}$ 分别定义为在数据逐次细分后，数据的偶—偶点和偶—奇点的离散傅里叶变换。

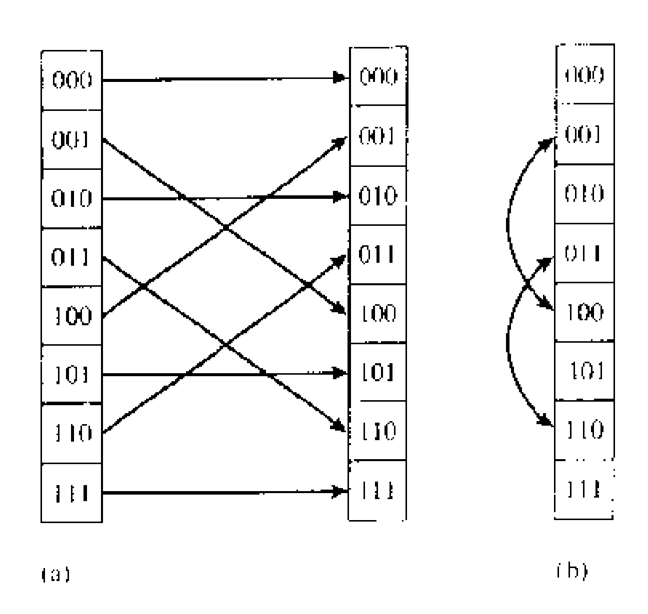
虽然有一些其它情况的处理方法，但最容易的情况还是，原始 $N$ 为 2 的整幂次项的 FFT。如果数据集的界长不是 2 的幂次时，则可添上一些零值，直到界长 2 的下一幂次。(在随后的章节中，将给一些更复杂的建议。)有了这种对 $N$ 的限制，显然我们可以继续应用尼尔森-兰佐斯引理，直到我们将全部数据细分成界长为 1 的变换。什么是界长为 1 的傅里叶变换呢？它正是把一个输入数复制成它的一个输出值的恒等运算！换言之，对于所有 $\log_2 N$ 个 $e$ 和 $o$ 的每个模式，都有一个一点变换，它正是输入数字中的一个 $f_n$

$$
F_k^{eoeeoeo\cdots oee} = f_n \qquad \text{对于某个 } n \tag{12.2.4}
$$

(当然，这个一点变换事实上不依赖于 $k$，因为它对 k 的循环周期为1。)

下一步策略是，断定哪个 $n$ 值对应于式(12.2.4)中哪一个 $e$ 和 $o$ 的模式。答案是：将 $e$ 和 $o$ 模式的次序颠倒，然后令 $e=0$，$o=1$，则得到 $n$ 值的二进制数。能明白为什么这样做吗？因为将数据逐次细分偶数和奇数就是逐次检验 $n$ 中的低位比特位(最低有效位)。这个**位序颠倒**思想与丹尼尔森-兰佐斯引理一起可以开拓成一种更灵巧的方法，使 FFT 更为实用：假如我们将数据 $f_j$ 的初始向量，把它重排成位序颠倒的顺序(见图12.2.1)，以致使每个数字不再按原 $j$ 的顺序排列，而是按 j 的二进制数颠倒后得出的数排列。于是，丹尼尔森-兰佐斯引理的递归应用的内部操作就变得十分简单，所给的点是一点变换。我们组合相邻的一对点就得到两点变换，再组合这些两点变换对的相邻对，又得到 4 点变换，以此类推，直到整个数据集的一半和另一半组合成最终的变换。每组合一次需 $N$ 阶运算，并且明显地有 $\log_2 N$ 次组合，因此整个算法是 $N\log_2 N$ 阶运算量。(假设分类成位序颠倒顺序的过程，其运算量不

大于 $N\log_2N$ 阶，通常情况也正是如此。)




(a)两个数组之间的对应关系，它相当于(b)数组中适当位置上重新排列。位序颠倒重新排序是快速傅里叶变换算法(FFT)的必须部分。

图12.2.1 有位序颠倒重新排序一数组(界长为8)

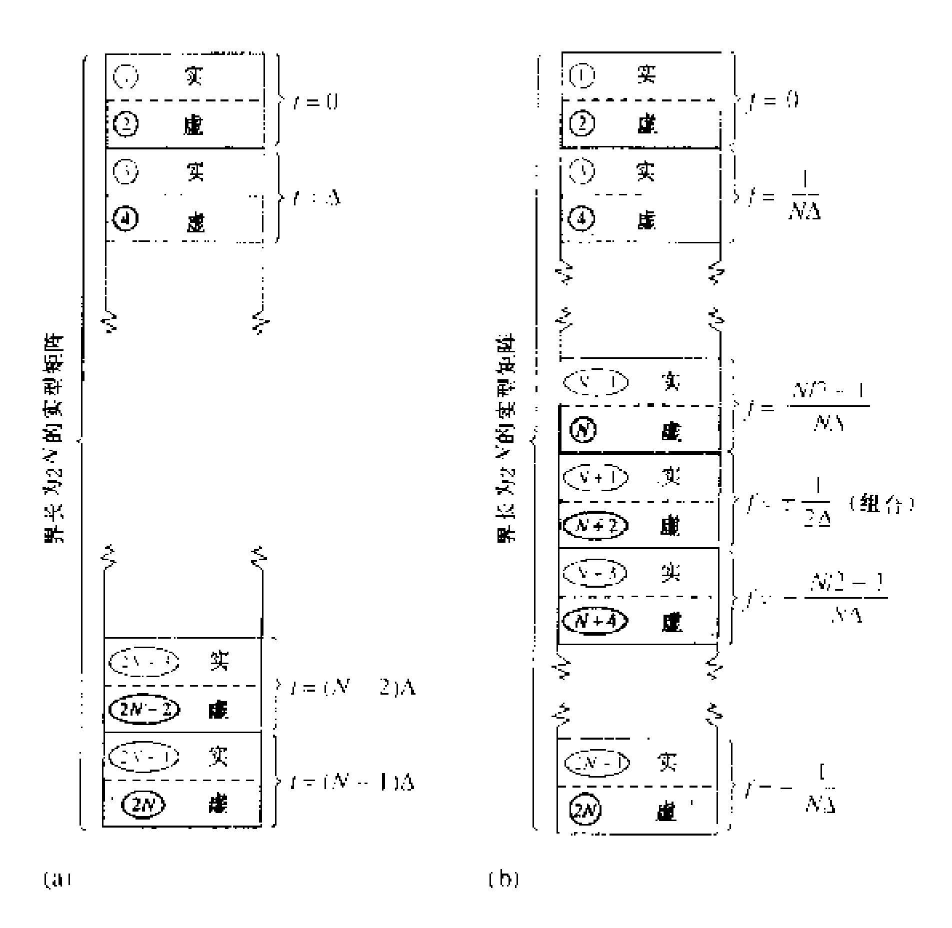
于是，FFT 算法的结构分为两个部分。第一部分，将数据整理成位序颠倒的次序。幸运的是，这并不占据更多的存储空间，因为它只包含元素的交换。(若 $k_1$ 和 $k_2$ 的位序颠倒，则 $k_2$ 亦是 $k_1$ 的位序颠倒。)第二部分，有一个执行 $\log_2N$ 次的外循环，它依次计算界长为 2，4，8，...，$N$ 的变换。对于这一过程的每一步来说，为了履行丹尼尔森-兰佐斯引理，有两个嵌套的内循环，其涉及到已计算的子变换和每个变换的元素。通过限制外部调用正弦和余弦到外层循环，可以使运算更有效，在外层循环中只调用它们 $\log_2N$ 次。倍角的正弦和余弦的计算是通过内循环中简单的递归关系进行的(参考式(5.5.6))。

下面给出的 FFT 算法程序是以布伦纳(N. M. Brenner)原来写的一个程序为基础。输入量是：复数据点的个数(nn)、数据数组(data[1..2 * nn]) 和 isign，isign 必须置成 +1 或 -1，它是式(12.1.7)中指数上 i 符号。当 isign 置写 -1，则程序是计算逆变换式(12.1.9)——只是程序中没有乘上等式中出现的归一化因子$1/N$，用户可以自己补加上。

注意，变量 nn 是复数据点的个数。实型数组(data[1..2 * nn]的实际界长是2倍 nn，而每个复数值占据了两个相继的存储单元。换言之，data[1]是 $f_0$的实部，data[2]是 $f_0$的虚部，以此类推，直到 data[2 * nn-1]是 $f_{N-1}$的实部，data[2 * nn]是 $f_{N-1}$的虚部。对于 nn 个复数，FFT 程序返回以同样方式压缩的 $F_n$。

零频率分量 $F_0$的实部和虚部在 data[1]和 data[2]中；最小的非零正频率的实部和虚部在 dat[3]和 data[4]中"；(数值上)最小的非零负频率的实部和虚部在 data[2 * nn-1]和 data[2 * nn]中。数量上增长的正频率存储在实-虚对 data[5]和 data[6]，直到 data[nn-1]和 data[nn]中。数量上增长的负频率存储在 data[2 * nn-3]和 data[2 * nn-2]直到 data[nn+3]和 d[nn+4]之中。最后，这对 data[nn-1]和 data[nn+2]存储一个混叠点的实部

和虚，这混叠点包含最大正频率和最小负频率。读者应该试着熟悉复频率谱的这种内存排列方式，如图12.2.2所示，因为这是一种实用标准排列。



(a)输入数组包含 $N$(2的幂次)个时间上的变样本值，形成界长为$2N$ 的实型数组，其实部和虚部交替布置。(b)输出数组是包含 $N$ 个频率值上的复傅里叶频谱，实部和虚部也交替相隔，数组从零频率开始，逐渐排列到最大正频率(这一处也可以作为最小的负频率)，然后紧着是负频率，从次最小的负频率一直升到恰好位于零以下的负频率

图12.2.2 FFT 的输入和输出数组

这里要提醒读者，即使输入数组从零点起始，即范围为 data[0...2 * nn －1]，仍可以使用这 four1程序而不作修改。在这种情况中，当 **four1** 被调用时只须简单地将 data 的指针减1，即调用 **four1**(data－1,1024,1)。这时 $f_0$ 的实部将返回到 data[0]，而虚部返回到 data[1]，以此类推。见第1.2节。

```
#include <math.h>
#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr

void four1(float data[], unsigned long nn, int isign)
    假如isign 输入为+1，则置换后的 data[1...2 * nn]是输入的离散傅里叶变换；或者假如 isign 输入为－1，则置换后的 data[1...2 * nn]是输入的 nn 个离散傅里叶的逆变换。data 是界长为 nn 的复数组或者等价于界长为2 * nn 的实数组，nn 必须是2的整数幂(这点未检验!)。
{
    unsigned long n,mmax,m,j,istep,i;
    double wtemp,wr,wpr,wpi,wi,theta;          关于三角递归的双精度
    float tempr,tempi;
```

```
    n=nn << 1;
    j=1;
    for (i=1;i<n;i+=2) {                          这是位序颠倒部分的程序
        if (j > i) {
            SWAP(data[j],data[i]);                交换两个复数
            SWAP(data[j+1],data[i+1]);
        }
        m=n >> 1;
        while (m >= 2 && j > m) {
            j -= m;
            m >>= 1;
        }
        j += m;
    }
    下面开始是程序的丹尼尔森-兰佐斯部分
    mmax=2;
    while (n > mmax) {                            执行log2 nn次外循环
        istep=mmax << 1;
        theta=isign*(6.28318530717959/mmax);      关于三角递归的初始赋值
        wtemp=sin(0.5*theta);
        wpr = -2.0*wtemp*wtemp;
        wpi=sin(theta);
        wr=1.0;
        wi=0.0;
        for (m=1;m<mmax;m+=2) {                   这是两个嵌套内循环
            for (i=m;i<=n;i+=istep) {
                j=i+mmax;                         这是丹尼尔森-兰佐斯公式
                tempr=wr*data[j]-wi*data[j+1];
                tempi=wr*data[j+1]+wi*data[j];
                data[j]=data[i]-tempr;
                data[j+1]=data[i+1]-tempi;
                data[i] += tempr;
                data[i+1] += tempi;
            }
            wr=(wtemp=wr)*wpr-wi*wpi+wr;          三角递归
            wi=wi*wpr+wtemp*wpi+wi;
        }
        mmax=istep;
    }
}
```

（程序 **four1** 的双精度版本称为 **dfour1**，它用于第20.6节的程序 **mpmul** 中。读者可很容易地进行这种改变，或者也可以从数值程序集软盘中获得改变好的双精度的程序。）

### 12.2.1 其它 FFT 算法

我们应该提到，对于上述给出的基本 FFT 算法还有一些变形算法。正如我们已经看到，上述算法，首先将输入元素重排成位序颠倒的次序，然后用 $\log_2 N$ 个迭代建起输出变换。在文献中，这种算法称为**时间抽选 FFT 算法**或**库利-图凯**(Cooley-Tukey)**FFT 算法**。现还可以导出另一种 FFT 算法，它首先对输入数据经过一系列 $\log_2 N$ 次迭代，然后将输出值重排成位序颠倒的次序。这种算法称为**频率抽选 FFT 算法**或**桑德-图凯**(Sande-Tukey)**FFT 算法**。在一些应用中，例如卷积(第13.1节)来说，人们先将数据集放入傅里叶域内，然后经过某些操作后，再返回来。在这情况下，可以避免所有的位序颠倒。使用频率抽选算法（不用位序颠倒）进入"不规则的"傅里叶域内，在那里进行操作，然后用逆算法（也不用位序颠倒）再返回到时域。虽然，这样处理在理论上看是很完美的，但实际上节省不了很多的运算时间，因为位序颠倒只呈现 *FFT* 运算量的一小部分，并且在频率域中，大多数有用的运算都要求知道哪

些点对应于哪些频率。

另一类 FFT 算法并不把界长为 $N$ 的初始数据集细分成界长为 1 的基本变换，而是细分成某些 2 较小幂次，例如 $N=4$，基数为 4 的 FFT；或 $N=8$，基数为 8 的 FFT。于是，这些小的变换可用最优化编码的小段来完成，这些最优化编码是利用了这种特别小的 $N$ 之独特的对称性。例如，当 $N=4$ 时，所参与的三角正弦和余弦全部是 ±1 或 0，所以许多乘法可省略，只剩下大量的加法和减法。这方法能比简单的 FFT 算法快 20% 或 30%，这是很有意义的，但并不是压倒一切的优越。

对于界长为 $N$ 而 $N$ 不是 2 的幂次的数据集，也有一些 FFT 算法。它们是通过使用类似于尼尔森—兰佐斯引理的方法，将初始问题逐次细分成一些较小的问题，这时不是每次一半一半地细分，而是每次用可除尽 $N$ 某个最小的素数因子去细分。$N$ 的最大的素数因子越大，这种方法的效果越差。如果 $N$ 本身是素数，则不可能再细分，那么用户（不管他是否知道）正采用着慢速的傅里叶变换，它是 $N^2$ 阶，而不是 $N\log_2N$ 阶运算量。我们建议这种 FFT 方法留待以后去弄清楚。然而也许有一些例外，即威努格特（Winograd）傅里叶变换算法。威努格特算法在某些方面是类似于基4和基8的 FFT 算法，威努格特已推导出高度优化编码可用于小的 $N$，例如 $N=2,3,4,5,7,8,11,13,16$ 等，进行离散傅里叶变换。这种算法采用了新颖而巧妙的方法组合子因子。在分层次处理之前和其之后，这一方法都包含数据的重新排序，但是它允许在算法内对乘法次数作有效归并。对于一些特别的 $N$ 值，威努格特算法可以比最靠近 2 的整数幂的简单 FFT 算法要有效地快（例如，快一倍）。然而，速度上的优势，必须与在这些变换中包含相当复杂的数据变址以及威努格特变换不能够"同址"计算的事实进行权衡比较。

最后，一类有趣的快速求卷积的变换是数论变换。这些结构是用某个大素数 $N+1$ 为模的整数算法来代替1的 $N$ 次根。严格地说，这已根本不是傅里叶变换，但是它们性能十分相似，而且运算速度快得多，另一方面，它们的用途象相关和卷积一样受到一些限制，这因为变换本身已不容易解释为"频率"谱了。

参考文献和进一步读物：


Brigham, E. O. 1974, *The Fast Foruier Transform* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Van Loan, C. 1992, *Computational Frameworks for the Fast Fourier Transform* (Philadelphia: S. I. A. M.).

Beauchamp, K. G. 1984, *Applications of Walsh Functions and Related Functions* (New York: Academic Press) [non-Fourier treansforms].


## 12.3 实函数的 FFT、正弦变换和余弦变换

常见的是这种情况，即期望求 FFT 的数据由实值样本 $f_j$ $(j=0,\ldots,N-1)$ 组成。为了使用 **four1**，我们将这些样本放入一个复型数组中，并令其所有虚部为零。计算所得的变换 $F_n$ $(n=0,\ldots,N-1)$ 满足 $F_{N-n}^*=F_n$。因为对于 $F_0$、$F_{N/2}$ 及其它 $(N/2)-1$ 个独立值 $F_1\ldots F_{N/2-1}$ 来说，这个复值数组都是实值，故它与原来实数据集有相同的 $2(N/2-1)+2=N$ 个"自由度"。然而，从执行时间和对存储需求二者来看，对实型数据采用完全复数的 FFT 算法，其效率是很低的。读者可能会想到，还有一个更好的方法。

有两种更好的方法。第一种是"成批生产法"，将两个单独的实际数组合成输入数组，其组合方式是，能使各自单独的变换从计算结果中分离出来。这种方法在以下程序 **twofft** 中实施。这类似于搭配销售，购货者只需买一件物品时却被强迫买了两件。可是请记住，对于相关和卷积来说，这样可使两种函数的傅里叶变换都包括了，故这是一举两得的便利方法。第二种方法是，巧妙地将实型输入数组组合成界长只有一半的复型数组，而不要多余的零值，然后，可对这一组较短的界长执行复数型 FFT，这里有个诀窍就是如何从结果中得到所需要的答案，这在以下 **realft** 程序中实施。

## 12.3.1 两个实函数同时变换

首先，我们说明如何利用变换 $F_n$ 的对称性来一次处理两个实函数。因为输入数据 $f_j$ 是实型的，故离散傅里叶变换的分量满足

$$F_{N-n} = (F_n)^* \tag{12.3.1}$$

其中星号表示复共轭。同理，纯虚集合的 $g_j$ 的离散傅里叶变换有相反的对称性：

$$G_{N-n} = -(G_n)^* \tag{12.3.2}$$

因此通过压缩两个数据数组分别为 **four1** 复型输入数组的实部与虚部，我们可以同时求得界长均为 $N$ 的两个实函数的离散傅里叶变换，然后，借助于两种对称性，可以将计算结果的变换数组分离成两个复型数组。程序 **twofft** 实施了这些想法。

```
void twofft(float data1[], float data2[], float fft1[],
    float fft2[], unsigned long n)
    给定两个输入实数组data1[1..n]和data2[1..n]，本程序调用函数four1产生返回两个复型输出数组fft1和fft2，其每
    个复界长为n(即实维数为[1..2n])，它们包含各自的data的离散傅里叶变换。n必须是2的整数次幂。
{
    void four1(float data[], unsigned long nn, int isign);
    unsigned long nn3,nn2,jj,j;
    float rep,rem,aip,aim;

    nn3=1+(nn2=2+n+n);
    for (j=1,jj=2;j<=n;j++,jj+=2) {                 将两个实型数组组合成一个复型数组
        fft1[jj-1]=data1[j];
        fft1[jj]=data2[j];
    }
    four1(fft1,n,1);                                 求复型数组的变换
    fft2[1]=fft1[2];
    fft1[2]=fft2[2]=0.0;
    for (j=3;j<=n+1;j+=2) {
        rep=0.5*(fft1[j]+fft1[nn2-j]);               用对称性分离两个变换
        rem=0.5*(fft1[j]-fft1[nn2-j]);
        aip=0.5*(fft1[j+1]+fft1[nn3-j]);
        aim=0.5*(fft1[j+1]-fft1[nn3-j]);
        fft1[j]=rep;                                 将它们按两个复型数组输送出来
        fft1[j+1]=aim;
        fft1[nn2-j]=rep;
        fft1[nn3-j]= -aim;
        fft2[j]=aip;
        fft2[j+1]= -rem;
        fft2[nn2-j]=aip;
        fft2[nn3-j]=rem;
    }
}
```

逆过程又是怎样的呢?假设有两个复变换数组,每一个都具有式(12.3.1)的对称性,那么就可知道这两个变换的逆变换都是实函数。能否在单个的 FFT 中对两个变换求逆呢?这种求逆变换甚至比求变换更容易。可利用 FFT 是线性的这一事实,形成第一次变换加上 $i$ 倍第二次变换的总和。可用 **four1** 并设 isign = −1 来求逆。计算得到的复数组的实部和虚部就是两个所求的实函数。

### 12.3.2 单个实函数的 FFT

第二种方法允许我们不用冗余位便可执行单个函数的 FFT。为了实施这种方法,我们将这数据集一分为二,形成两个大小只有一半的实型数值。我们可以把上述程序用于这二组数组,当然结果将不是原始数据的变换。它将是两个变换的错乱集合,但其每个变换都有我们所需的一半信息。幸运的是,这种错乱是可处理的,作法如下:

正确划分原始数据的方法是,把偶数项的 $f_j$ 作为一个数据集,奇数项的 $f_j$ 作为另一数据集。这样做的巧妙之处是,可以选取原始实型数组,并把它当作一个界长只有一半的复型数 $h_j$ 来处理。第一个数据集是这复型数组的实部,第二个是它的虚部,如同 **twofft** 中规定的一样,而不需要重新组合安排。换言之,$h_j = f_{2j} + if_{2j+1}(j=0,\ldots,N/2-1)$。我们将这数组提交给 **four1**,它将返回一个复型数组 $H_n = F_n^e + iF_n^o(n=0,\ldots,N/2-1)$,并且

$$F_n^e = \sum_{k=0}^{N/2-1} f_{2k} e^{2\pi ikn/(N/2)}$$

$$F_n^o = \sum_{k=0}^{N/2-1} f_{2k+1} e^{2\pi ikn/(N/2)} \tag{12.3.3}$$

程序 **twofft** 的讨论是告诉读者如何从 $H_n$ 中分出两个变换 $F_n^e$ 和 $F_n^o$。现在,如何做才能将它们变成原始数据集 $f_j$ 的变换 $F_n$ 呢?我们建议回顾一下式(12.2.3):

$$F_n = F_n^e - e^{2\pi in/N} F_n^o \qquad n = 0,\ldots,N-1 \tag{12.3.4}$$

根据我们的实型数据集(冒充复型数组)的变换 $H_n$ 直接来表达,其结果是

$$F_n = \frac{1}{2}(H_n + H_{N/2-n}^*) - \frac{i}{2}(H_n - H_{N/2-n}^*)e^{2\pi in/N} \qquad n=0,\ldots,N-1 \tag{12.3.5}$$

几点注意事项:

- 由于 $F_{N-n}^* = F_n$,所以没有存储整个频谱的点。正半频已足够,并能如同原始数据一样存入同一数组中。事实上,这一运算可以同址完成。
- 即使如此,我们仍需要数值 $H_n$, $n=0,\ldots,N/2$,而 **four1** 只给出数值 $n=0,\ldots,N/2-1$。利用对称性可补救,并有 $H_{N/2}=H_0$。
- 数值 $F_0$ 和 $F_{N/2}$ 都是实数且独立。为了确实把整个 $F_n$ 插入原始数组空间,把 $F_{N/2}$ 置入 $F_0$ 的虚部是比较合适的。
- 尽管形式复杂,但以上过程是可逆的,首先从 $F_0$ 中脱离出 $F_{N/2}$,然后构成

$$F_n^e = \frac{1}{2}(F_n + F_{N/2-n}^*)$$

$$F_n^o = \frac{1}{2}e^{-2\pi in/N}(F_n - F_{N/2-n}^*) \quad n = 0,\ldots,N/2-1 \tag{12.3.6}$$

再用 **four1** 来寻求 $H_n = F_n^{(1)} + iF_n^{(2)}$ 的逆变换。令人惊奇的是,具体的代数步骤实际上与

正向变换的那些步骤完全相同。

下面是相应的典型程序：

```
#include <math.h>

void realft (float data[], unsigned long n, int isign)
    计算一组n个实值数据点的傅里叶变换。用复傅里叶变换的正半频率替换这些数据(它存储在数组data[1..n]中)。
    复变换的第一个和最后一个分量的实数值分别返回单元data[1]和data[2]中。n必须是2的幂次。这个程序也能计
    算复数据数组的逆变换，只要该数组是实值数据的变换(在这种情况下，其结果必须乘以2/n)即可。
{
    void four1(float data[], unsigned long nn, int isign);
    unsigned long i,i1,i2,i3,i4,np3;
    float c1=0.5,c2,h1r,h1i,h2r,h2i;
    double wr,wi,wpr,wpi,wtemp,theta;                 为三角递归而设双精度

    theta=3.141592653589793/(double) (n>>1);          递归的初始赋值
    if (isign == 1) {
        c2 = -0.5;
        four1(data,n>>1,1);                           此处是正向变换
    } else {
        c2=0.5;                                       否则设置逆变换
        theta = -theta;
    }
    wtemp=sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi=sin(theta);
    wr=1.0+wpr;
    wi=wpi;
    np3=n+3;
    for (i=2;i<=(n>>2);i++) {                         情况i=1以下分别完成
        i4=1+(i3=np3-(i2=1+(i1=i+i-1)));
        h1r=c1*(data[i1]+data[i3]);                   两个分离变换是从data中分离出来
        h1i=c1*(data[i2]-data[i4]);
        h2r = -c2*(data[i2]+data[i4]);
        h2i=c2*(data[i1]-data[i3]);
        data[i1]=h1r+wr*h2r-wi*h2i;                   此处重新组合以形成原始实型数据的真
        data[i2]=h1i+wr*h2i+wi*h2r;                       实变换
        data[i3]=h1r-wr*h2r+wi*h2i;
        data[i4] = -h1i+wr*h2i+wi*h2r;
        wr=(wtemp=wr)*wpr-wi*wpi+wr;                  递归式
        wi=wi*wpr+wtemp*wpi+wi;
    }
    if (isign == 1) {
        data[1] = (h1r=data[1])+data[2];              同时挤压第一个和最后一个数据使它们
        data[2] = h1r-data[2];                            都在原始数组中
    } else {
        data[1]=c1*((h1r=data[1])+data[2]);
        data[2]=c1*(h1r-data[2]);
        four1(data,n>>1,-1);                          此处是isign=-1的逆变换
    }
}
```

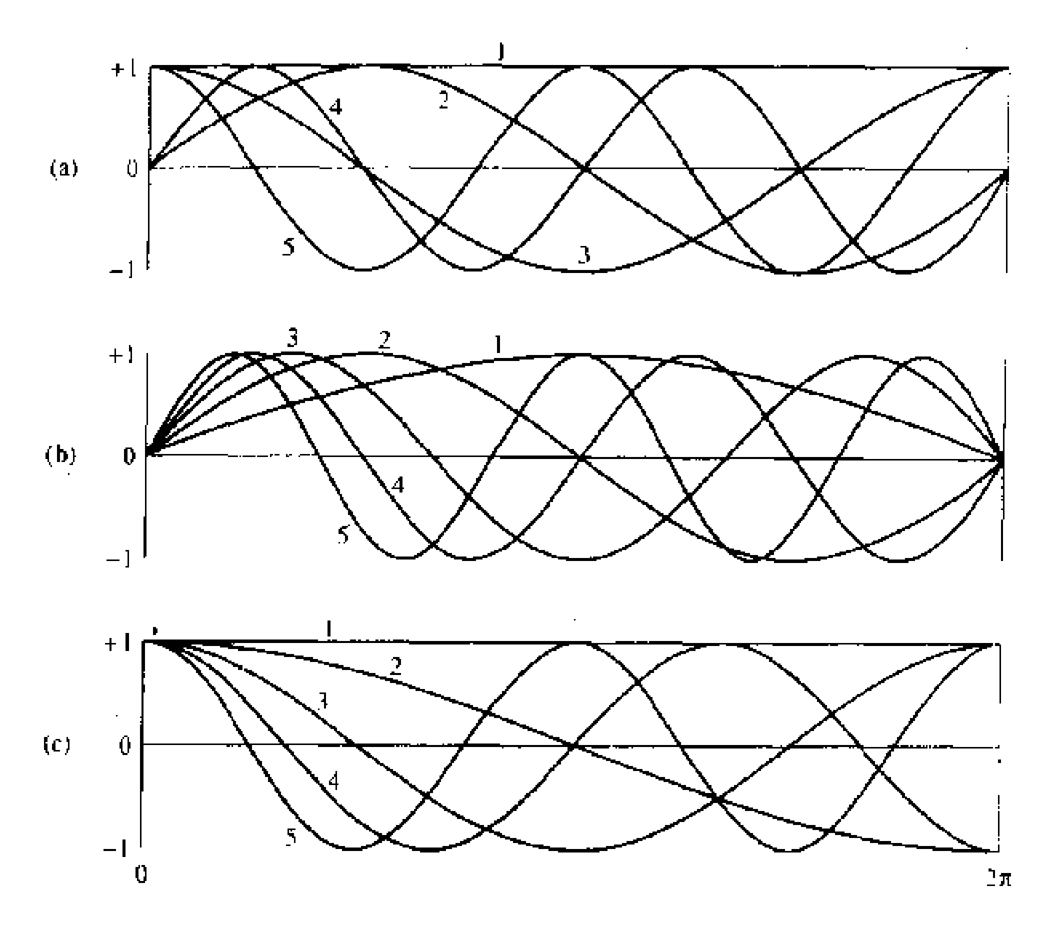
## 12.3.3 快速正弦和余弦变换

在函数的傅里叶变换的其它用途中，还可用来解微分方程(参阅第19.4节)。求解的最常用边界条件是：1) 它们在边界上取零值，或者 2) 在边界上它们导数为零。在第一种情况下，常用的变换是**正弦变换**，给出如下：

$$F_k = \sum_{j=1}^{N-1} f_j \sin(\pi jk/N) \qquad \text{正弦变换} \tag{12.3.7}$$

其中 $f_j(j=0,\ldots,N-1)$是数据数组，且 $f_0=0$。

乍一看，上式好象就是离散傅里叶变换的虚部。可是，正弦的变量与用离散傅里叶变换来计算的变量值相差 2 倍。正弦变换只使用在 0 至 $2\pi$ 区间内一个完全函数集的一些**正弦函数**，并且如同我们将看到的，余弦变换也只使用一些**余弦函数**。相比之下，标准的 FFT 却使用了正弦和余弦两个函数，并且只使用各函数集中的一半(见图12.3.1)。




绘制了傅里叶变换(a)、正弦变换(b)和余弦变换(c)所使用的基函数。每种情况展示了前五个基函数(对于傅里叶变换，基函数的实部和虚部都予以展示)。虽然相同的基函数发生在不止一个的变换中，但基函数集是不同的。例如，正弦变换中标号为(1)、(3)、(5)的基函数不出现在傅里叶变换中。在表明的区间内，任意函数可由三个基函数集的任一函数集展开。但是，若用正弦变换和余弦变换展开函数，最好函数与各自基函数的边界条件相匹配，即零函数值为正弦变换，零导数值为余弦变换。

图12.3.1 快速正弦和余弦变换

可以使表达式(12.3.7)"强制适合"于一种形式，这种形式可允许用 FFT 来计算。这一思想是，通过函数的最后列表值来扩展给定函数的右端。我们将数据的界长扩大一倍，其方式是使它们成为关于 $j=N$ 的奇函数，此时 $f_N=0$。

$$f_{2N-j} \equiv -f_j \qquad j=0,\ldots,N-1 \tag{12.3.8}$$

考虑这个扩展函数的 FFT，

$$F_k = \sum_{j=0}^{2N-1} f_j e^{2\pi ijk/(2N)} \tag{12.3.9}$$

该和式的一半，从 $j=N$ 到 $j=2N-1$，可用 $j'=2N-j$ 重写：

$$\begin{aligned}\sum_{j=N}^{2N-1} f_j e^{2\pi ijk/(2N)} &= \sum_{j'=1}^{N} f_{2N-j'} e^{2\pi i(2N-j')k/(2N)} \\ &= -\sum_{j'=0}^{N-1} f_{j'} e^{-2\pi ij'k/(2N)}\end{aligned} \tag{12.3.10}$$

所以

$$\begin{aligned}F_k &= \sum_{j=0}^{N-1} f_j \left[e^{2\pi ijk/(2N)} - e^{-2\pi ijk/(2N)}\right] \\ &= 2i\sum_{j=0}^{N-1} f_j \sin(\pi jk/N)\end{aligned} \tag{12.3.11}$$

因此，除了因子 $2i$ 外，我们从扩展函数的 FFT 中得到正弦变换。

这种方法由于扩展了数据，使得计算效率降低了一半。这种低效率呈现在 FFT 的输出，因为变换的每个元素的实部都是零。对于一维问题，效率降低一半尚能接收，特别是考虑到这一方法的简单性。但当我们计算二维或三维的偏微分方程时，效率将降低到原来的1/4或1/8，所以应努力消除这种低效性。

从原始实型数据数组 $f_j$ 中，我们可以构造一个辅助数组 $y_j$，并将它运用于程序 **realft** 中。然后，可用输出来构成我们所求的变换。对于数据 $f_j(j=1,\ldots,N-1)$的正弦变换，辅助数组为

$$\begin{aligned}y_0 &= 0 \\ y_j &= \sin(j\pi/N)(f_j + f_{N-j}) + \frac{1}{2}(f_j - f_{N-j}) \quad j=1,\ldots,N-1\end{aligned} \tag{12.3.12}$$

这一数组与原始数组的维数相同。注意，第一项关于 $j=N/2$是对称的，且第二项是反对称的。因而当 **realft** 用于 $y_j$ 时，其结果具有下式给出的实部 $R_k$ 和虚部 $I_k$：

$$\begin{aligned}R_k &= \sum_{j=0}^{N-1} y_j \cos(2\pi jk/N) \\ &= \sum_{j=1}^{N-1} (f_j + f_{N-j})\sin(j\pi/N)\cos(2\pi jk/N) \\ &= \sum_{j=0}^{N-1} 2f_j \sin(j\pi/N)\cos(2\pi jk/N) \\ &= \sum_{j=0}^{N-1} f_j\left[\sin\frac{(2k+1)j\pi}{N} - \sin\frac{(2k-1)j\pi}{N}\right] \\ &= F_{2k+1} - F_{2k-1}\end{aligned} \tag{12.3.13}$$

$$\begin{aligned}I_k &= \sum_{j=0}^{N-1} y_j \sin(2\pi jk/N) \\ &= \sum_{j=1}^{N-1} (f_j - f_{N-j})\frac{1}{2}\sin(2\pi jk/N) \\ &= \sum_{j=0}^{N-1} f_j \sin(2\pi jk/N)\end{aligned}$$

$$= F_{2k} \qquad (12.3.14)$$

所以，$F_k$ 可以确定如下：

$$F_{2k} = I_k \qquad F_{2k+1} = F_{2k-1} + R_k \quad k = 0,\ldots,(N/2-1) \qquad (12.3.15)$$

$F_k$ 的偶数项可以直接确定。而奇数项需要递推，在式(12.3.15)中设 $k=0$，利用 $F_1=-F_{-1}$ 求得

$$F_1 = \frac{1}{2}R_0 \qquad (12.3.16)$$

实用程序为：

```
include <math.h>

void sinft (float y[], int n)
    计算存储在数组 y[1..n]中一组 n 个实值数据点的正弦变换。n 必须是2的幂次。在出口 y 被它的变换所代替。这个
    程序，不加任何改变也可计算正弦变换的逆变换，但在这种情况下，输出数组应乘以2/n。
{
    void realft(float data[], unsigned long n, int isign);
    int j,n2=n+2;
    float sum,y1,y2;
    double theta,wi=0.0,wr=1.0,wpi,wpr,wtemp;          为三角递归而设双精度

    theta=3.14159265358979/(double) n;                 递归的初始赋值
    wtemp=sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi=sin(theta);
    y[1]=0.0;
    for (j=2;j<=(n>>1)+1;j++) {
        wr=(wtemp=wr)*wpr-wi*wpi+wr;     计算辅助数据的正弦
        wi=wi*wpr+wtemp*wpi+wi;          为继续递归需要余弦
        y1=wi*(y[j]+y[n2-j]);            构造辅助数组
        y2=0.5*(y[j]-y[n2-j]);
        y[j]=y1+y2;                      j 和N-j 项有关联
        y[n2-j]=y1-y2;
    }
    realft(y,n,1);                       求辅助数组的变换
    y[1]*=0.5;                           对以下奇数项的和式初始赋值
    sum=y[2]=0.0;
    for (j=1;j<=n-1;j+=2) {
        sum += y[j];
        y[j]=y[j+1];                     变换中偶数项直接确定
        y[j+1]=sum;                      通过运行和式奇数项被确定
    }
}
```

有趣的是，正弦变换亦是其自身的逆变换。如读者应用正弦变换两次，便可得到原始数组，但只是乘上了一个因子 $N/2$。

微分方程另一种常见的边界条件是，边界上函数的导数为零。在这种情况下，正常的变换是**余弦变换**。定义这种变换有几种可能的方式。每种定义都可认为是一种结果，该结果可以通过不同的扩展方法，从给定的数组构造一个双倍界长的偶数组来获得，或者也可以用所扩展的数组中是否包括 $2N-1$、$2N$ 或其它一些点的数来求得。实际上，许许多多可能的形式中只有两种是有用的，所以我们只讨论这两种形式。

余弦变换的第一种形式用 N+1 个数据点表示为：

$$F_k = \frac{1}{2}[f_0 + (-1)^k f_N] + \sum_{j=1}^{N-1} f_j \cos(\pi jk/N) \qquad (12.3.17)$$

该结果是将给定数据扩展成为关于 $j=N$ 的偶数组而获得的，并有

$$f_{2N-j}=f_j,\quad j=0,\dots,N-1 \tag{12.3.18}$$

若将扩展数组代入式(12.3.9)，接着类似地按推导式(12.3.11)的步骤，就将发现这傅里叶变换是余弦变换式(12.3.17)的两倍。关于式(12.3.17)另一种方法正是前面已提到的切比雪夫、高斯-洛巴托求积公式(参见第4.5节)，它常应用于克伦肖-柯蒂斯积分(Clenshaw Curtis)(参见第5.9节公式(5.9.4))。

另外一种方法，可以计算这个变换而不降低一半效率。此时，辅助函数是

$$y_j=\frac{1}{2}(f_j+f_{N-j})-\sin(j\pi/N)(f_j-f_{N-j})\qquad j=0,\dots,N-1 \tag{12.3.19}$$

替换公式(12.3.15)，现在程序 **realft** 给出

$$F_{2k}=R_k\quad F_{2k+1}=F_{2k-1}+I_k\qquad k=0,\dots,(N/2-1) \tag{12.3.20}$$

在此情况下，对于奇数 $k$ 递归的起始值是

$$F_1=\frac{1}{2}(f_0-f_N)+\sum_{j=1}^{N-1}f_j\cos(j\pi/N) \tag{12.3.21}$$

该和式中没有出现 $R_k$ 和 $I_k$，所以可以在数组 $y_j$ 生成过程中累计该和式。

这个变换也是其自身的逆变换，所以下述程序能用于求变换及逆变换。请注意，虽然该余弦变换的形式有 $N+1$ 个输入和输出，但它仅仅传送界长为 N 的数组到程序 **realft** 中去。

```
#include <math.h>
#define PI 3.141592653589793

void cosft1(float y[], int n)
    计算实值数据点的一组y[1..n]的余弦变换，所得变换数据替代了数组y中的原始数据。n必须是2的幂次 该程序
    不用改变就可以计算余弦变换，但此时输出数组必须乘上2/n。
{
    void realft(float data[], unsigned long n, int isign);
    int j,n2;
    float sum,y1,y2;
    double theta,wi=0.0,wpi,wpr,wr=1.0,wtemp;          对三角递归需双精度

    theta=PI/n;                                       递归的初始赋值
    wtemp=sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi=sin(theta);
    sum=0.5*(y[1]-y[n+1]);
    y[1]=0.5*(y[1]+y[n+1]);
    n2=n+2;
    for (j=2;j<=(n>>1);j++) {                         因为y[n/2+1]不变，所以不必使j=n/2+1
        wr=(wtemp=wr)*wpr-wi*wpi+wr;                  执行递归
        wi=wi*wpr+wtemp*wpi+wi;
        y1=0.5*(y[j]+y[n2-j]);                        计算辅助函数
        y2=(y[j]-y[n2-j]);
        y[j]=y1-wi*y2;                                j和N-j项有关联
        y[n2-j]=y1+wi*y2;
        sum += wr*y2;                                 为了用于后面所呈现的变换而求和
    }
    realft(y,n,1);                                    计算辅助函数的变换
    y[n+1]=y[2];
    y[2]=sum;                                         sum是式(12.3.21)中的F1值
    for (j=4;j<=n;j+=2) {                             式(12.3.20)
        sum += y[j];
        y[j]=sum;
```

余弦变换的第二种重要形式定义为

$$F_k = \sum_{j=0}^{N-1} f_j \cos\frac{\pi k(j+\frac{1}{2})}{N} \tag{12.3.22}$$

其逆变换

$$f_j = \frac{2}{N}\sum_{k=0}^{N-1}{}' F_k \cos\frac{\pi k(j+\frac{1}{2})}{N} \tag{12.3.23}$$

这里,和式上的符号"′"表示对 $k=0$ 的那项有一系数1/2在前面。这一形式可以由扩展给定的数据来呈现,定义扩展的数据从 $j=0,\ldots,N-1$ 到 $j=N,\ldots,2N-1$,在这种情况下,它对点 $N-\frac{1}{2}$ 是偶的和周期的(因此关于 $j=-\frac{1}{2}$ 也是偶的)。这种形式(12.3.23)与高斯-切比雪夫求积有关(参见式(4.5.19)),也与切比雪夫近似(第5.8节,式(5.8.7))和克伦肖-柯蒂斯积分(第5.9节)有关。

在"交错"网格上解微分方程时,余弦变换的这种形式是有效的,其中在网格点之间的中途上变量被置于中心。它也是数据压缩和图像处理领域中的标准形式。

在本情况下,使用的辅助函数类似于式(12.3.19):

$$y_j = \frac{1}{2}(f_j + f_{N-j-1}) + \sin\frac{\pi(j+\frac{1}{2})}{N}(f_j - f_{N-j-1}) \quad j=0,\ldots,N-1 \tag{12.3.24}$$

执行类似于式(12.3.12)至式(12.3.15)所采用的步骤,可推导得

$$F_{2k} = \cos\frac{\pi k}{N}R_k - \sin\frac{\pi k}{N}I_k \tag{12.3.25}$$

$$F_{2k-1} = \sin\frac{\pi k}{N}R_k + \cos\frac{\pi k}{N}I_k + F_{2k+1} \tag{12.3.26}$$

注意由方程(12.3.26)得:

$$F_{N-1} = \frac{1}{2}R_{N/2} \tag{12.3.27}$$

所以偶分量可直接从式(12.3.25)求得,而奇分量可由式(12.3.27)起始,从 $k=N/2-1$ 逐渐递推式(12.3.26)求得。

因为这变换不是其自身的反变换,所以我们必须颠倒上述步骤而求出逆变换。实际程序如下:

```
#include <math.h>
#define PI 3.141592653589793

void cosft2(float y[], int n, int isign)
计算实值数据的一维 y[1..n]的"交错"余弦变换。所得变换数据替换了数组 y 中原始数据。n 必须是2的幂次。令
isign=+1是余弦变换;而 isign=-1是逆余弦变换。在逆变换运算时,输出数组要乘上2/n。
{
    void realft(float data[], unsigned long n, int isign);
```

```
    int i;
    float sum,sum1,y1,y2,ytemp;
    double theta,wi=0.0,wi1,wpi,wpr,wr=1.0,wr1,wtemp;
    Double precision for the trigonometric recurrences.

    theta=0.5*PI/n;                                  递归的初始赋值
    wr1=cos(theta);
    wi1=sin(theta);
    wpr = -2.0*wi1*wi1;
    wpi=sin(2.0*theta);
    if (isign == 1) {                                正向变换
        for (i=1;i<=n/2;i++) {
            y1=0.5*(y[i]+y[n-i+1]);                  计算辅助函数
            y2=wi1*(y[i]-y[n-i+1]);
            y[i]=y1+y2;
            y[n-i+1]=y1-y2;
            wr1=(wtemp=wr1)*wpr-wi1*wpi+wr1;         执行递归
            wi1=wi1*wpr+wtemp*wpi+wi1;
        }
        realft(y,n,1);                               变换辅助函数
        for (i=3;i<=n;i+=2) {                        偶数项
            wr=(wtemp=wr)*wpr-wi*wpi+wr;
            wi=wi*wpr+wtemp*wpi+wi;
            y1=y[i]*wr-y[i+1]*wi;
            y2=y[i+1]*wr+y[i]*wi;
            y[i]=y1;
            y[i+1]=y2;
        }
        sum=0.5*y[2];                                用 1/2 R_{N/2} 对奇数项递归作初始赋值;
        for (i=n;i>=2;i-=2) {
            sum1=sum;                                对奇数项执行递归
            sum += y[i];
            y[i]=sum1;
        }
    } else if (isign == -1) {                        逆变换
        ytemp=y[n];
        for (i=n;i>=4;i-=2) y[i]=y[i-2]-y[i];        奇数项的差
        y[2]=2.0*ytemp;
        for (i=3;i<=n;i+=2) {                        计算 R_k 和 I_k
            wr=(wtemp=wr)*wpr-wi*wpi+wr;
            wi=wi*wpr+wtemp*wpi+wi;
            y1=y[i]*wr+y[i+1]*wi;
            y2=y[i+1]*wr-y[i]*wi;
            y[i]=y1;
            y[i+1]=y2;
        }
        realft(y,n,-1);
        for (i=1;i<=n/2;i++) {                       反转辅助数组
            y1=y[i]+y[n-i+1];
            y2=(0.5/wi1)*(y[i]-y[n-i+1]);
            y[i]=0.5*(y1+y2);
            y[n-i+1]=0.5*(y1-y2);
            wr1=(wtemp=wr1)*wpr-wi1*wpi+wr1;
            wi1=wi1*wpr+wtemp*wpi+wi1;
        }
    }
}
```

实施该算法的一种交替方法是,将 $f_j$ 的偶元素来复制到前 $N/2$个存储单元,而将 $f_j$ 的奇元素以倒序复制其次 $N/2$个存储单元,从而形成辅助函数。可是,如果没有暂存数组,实施这交替算法是不容易的,因此我们较喜欢用上述的同址算法。

最后,我们提到,对小的 $N$ 数存在着快速余弦变换,它不依赖辅助函数或不使用 FFT 程序,而是常常一些固定的小维数 $N$ 硬件编码后直接计算余弦变换。

参考文献和进一步读物：

Sorensen, H. V., Jones, D. L., Heideman, M. T., and Burrus, C. S. 1987, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-35, pp. 849~863.

Hou, H. S. 1987, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-35, pp. 1455-1461 [see for additional references].

Clarke, R. J. 1985, *Transform Coding of Images*, (Reading, MA; Addison-Wesley).

Gonzalez, R. J., and Wintz, P. 1987, *Digital Image Processing*, (Reading, MA: Addison-Wesley).

Chen, W., Smith, C. H., and Fralick, S. C. 1977, *IEEE Transactions of Communications*, vol. COM-25, pp. 1004~1009. [1]

## 12.4 二维或多维的 FFT

设有一个定义在二维网格 $0\leqslant k_1\leqslant N_1-1$，$0\leqslant k_2\leqslant N_2-1$ 的复函数 $h(k_1,k_2)$，我们可以在同样的网格上定义它的二维离散傅里叶变换是复函数 $H(n_1,n_2)$。

$$H(n_1,n_2)\equiv\sum_{k_2=0}^{N_2-1}\sum_{k_1=0}^{N_1-1}\exp(2\pi i k_2 n_2/N_2)\exp(2\pi i k_1 n_1/N_1)h(k_1,k_2) \tag{12.4.1}$$

将"下标为 2"的指数放到关于 $k_1$ 的和式之外，或者颠倒求和的次序，将"下标为 1"的指数放到关于 $k_2$ 的和式之外。我们可以发现，连续地在每个指标上计算原函数的一维 FFT 就能求得二维 FFT。用符号表示为

$$\begin{aligned}H(n_1,n_2)&=\text{指标 1 的 FFT(指标 2 的 FFT}[h(k_1,k_2)])\\&=\text{指标 2 的 FFT(指标 1 的 FFT}[h(k_1,k_2)])\end{aligned} \tag{12.4.2}$$

当然，为了实用，$N_1$ 和 $N_2$ 两者都应该是 FFT 的某个有效界长，通常均为 2 的幂次。用一维 FFT 的程序按照式(12.4.2)编制的二维 FFT 程序比最初表面上看起来要笨拙一点。因为一维的程序要求它输入数据按一维复型数组呈连续顺序，因此可发现，这样将从多维输入数组中无止境地复制数据出来，然后又将数据复制回去。因此这是不值得推荐的技术。当然，应该采用多维的 FFT 程序，下面我们给出一个程序。

显而易见，可将式(12.4.1)推广到二维以上。设 $L$ 维的一般表达式是

$$\begin{aligned}H(n_1,\ldots,n_L)\equiv\sum_{k_L=0}^{N_L-1}\cdots\sum_{k_1=0}^{N_1-1}&\exp(2\pi i k_L n_L/N_L)\times\cdots\\&\times\exp(2\pi i k_1 n_1/N_1)\,h(k_1,\ldots,k_L)\end{aligned} \tag{12.4.3}$$

其中 $n_1$ 和 $k_1$ 的范围从0至 $N_1-1$，…，$n_L$ 和 $k_L$ 的范围从0至 $N_L-1$。在式(12.4.3)中要调用多少次一维 FFT？相当多！首先对 $k_1,k_2,\ldots,k_{L-1}$ 的每个值作 FFT，求出 L 下标中的变换，然后对 $k_1,k_2,\ldots,k_{L-2}$ 和 $n_L$ 的每个值作 FFT，求出 $L-1$ 下标中的变换，以此类推。如果采用上述这种做法，就只有依靠某个已作过多次和全面程序操作的人了。

式(12.4.1)或(12.4.3)的逆变换正如期望的那样：将指数的 $i$ 变成 $-i$，并在整个式子前乘上一个整体因子 $1/(N_1\times\cdots\times N_L)$，多维 FFT 也具有类似一维情况中已讨论过的许多其它特性：

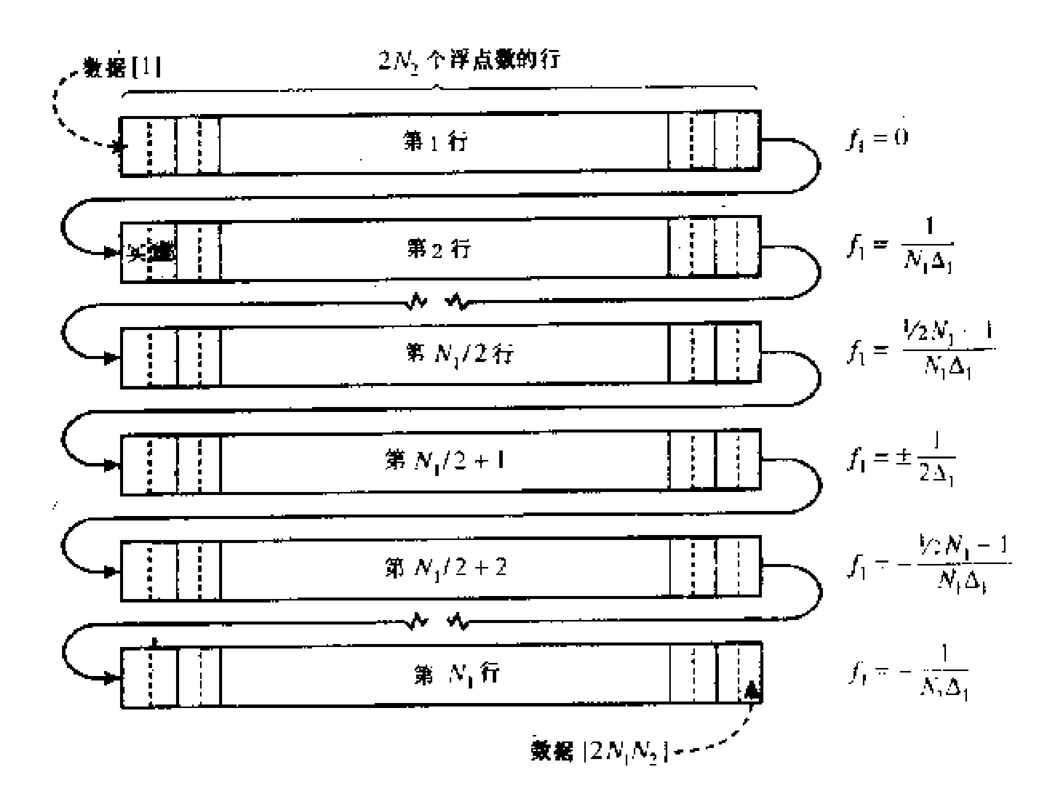
- 在变换中频率按环绕顺序安排，但现在是对每个单独的维数。
- 输入数据也按环绕顺序处理。假如经过这周期的鉴别(在任何维数)它们是不连续的，

则因为不连续性，在高频上频谱将会有某些虚假的功率。如果介意的话，处理方法就是移去多维的线性趋势。

- 若正要进行空间滤波，并且为环绕效应而烦恼，则需要围绕多维数组边界作零元填充。但是，请注意在多维变换中零元填充的代价有多大。如果使用了过多的零元填充，将耗费大量的存储，尤其是在三维和更高维的情况。
- 如果一维或多维的变换不存在足够的带宽限制的话，混叠常常会发生。

这里我们提供的程序 **fourn** 是根据布伦尔(N. M. Brenner)编写的一个程序派生出来的。作为输入，它要求：(i)一个标量，说明维数的个数，例如2；(ii)一个向量，说明每个维数的界长，例如(32,64)。请注意这些界长必须都是 2 的幂次，并且是在每个方向上复值的个数；(iii)一个通常等于 +1 的标量，它表示是求变换还是逆变换；最后(iv)数据数组。

关于数据数组说几句：程序 **fourn** 是以一个一维实数数组来存取数组，即 data[1..($2N_1N_2..N_L$)]，其界长是 $L$ 维界长乘积的两倍。假设数组表示为 $L$ 维复数组，其单个分量的顺序如下：(i)每一复数值使用两个连续的存储单元，实部随后跟虚部；(ii)当人们搜索数组时第一个下标变化最慢而最末一个下标变化最迅速(即在 C 标准语言中"以行存储")；(iii)下标变范围从 1 至它们的最大值(分别为 $N_1,N_2,...,N_L-1$)，而不是从 0 至 $N_1-1$，$N_2-1,...,N_L-1$。几乎所有使 **fourn** 工作失效的原因都是由于对数据数组的上述顺序不正确理解的结果。(图12.4.1说明了输出数据的格式。)




输入数据是二维 $N_1\times N_2$ 数组 $h(t_1,t_2)$(以复数的行存储)，输出以复数行存储，每行对应于特定的 $f_1$ 值，如图所示，每一行中频率 $f_2$ 的排列完全如图12.2.2所示，$\Delta_1$ 和 $\Delta_2$ 分别是沿1和2方向上的取样间隔。(实)数组元素的总数为 $2N_1N_2$，程序 **fourn** 也能用于高于二维的情况，并且以显而易见的方法推广存储的排列。

图12.4.1 二维 FFT 的输出 $H(f_1,f_2)$ 中频率的存储排列

```
#include <math.h>
#define SWAP(a,b) tempr=(a);(a)*(b);(b)=tempr

void fourn (float data[], unsigned nn[], int ndim, int isign)
```

用data的ndim维离散傅里叶变换替换它自身的data。若isign输入为1，nn[1..ndim]是包含每维界长(复值个数)的整型数组，他们必须全部是2的幂次。data是实型数组其界长为这些界长乘积的两倍，在这数组中数据以多维复型数组存储：每个元素的实部和虚部是以顺序放置，并且当人们沿data进行时，数组的最右边序号增加最迅速，对于二维数组，即等于以行存储数组。若isign输入为−1，则data被它们的逆变换乘以所有维数的界长之积所替代

```
{
    int idim;
    unsigned long i1,i2,i3,i2rev,i3rev,ip1,ip2,ip3,ifp1,ifp2;
    unsigned long ibit,k1,k2,n,nprev,nrem,ntot;
    float tempi,tempr;
    double theta,wi,wpi,wpr,wr,wtemp;          关于三角递推的双精度

    for (ntot=1,idim=1;idim<=ndim;idim++)      计算复值的总个数
        ntot *= nn[idim];
    nprev=1;
    for (idim=ndim;idim>=1;idim--) {           维数上的总循环
        n=nn[idim];
        nrem=ntot/(n*nprev);
        ip1=nprev << 1;
        ip2=ip1*n;
        ip3=ip2*nrem;
        i2rev=1;
        for (i2=1;i2<=ip2;i2+=ip1) {           这是程序的位序颠倒部分
            if (i2 < i2rev) {
                for (i1=i2;i1<=i2+ip1-2;i1+=2) {
                    for (i3=i1;i3<=ip3;i3+=ip2) {
                        i3rev=i2rev+i3-i2;
                        SWAP(data[i3],data[i3rev]);
                        SWAP(data[i3+1],data[i3rev+1]);
                    }
                }
            }
            ibit=ip2 >> 1;
            while (ibit >= ip1 && i2rev > ibit) {
                i2rev -= ibit;
                ibit >>= 1;
            }
            i2rev += ibit;
        }
        ifp1=ip1;                              这里开始程序的丹尼尔森—兰佐斯部分
        while (ifp1 < ip2) {
            ifp2=ifp1 << 1;
            theta=isign*6.28318530717959/(ifp2/ip1);    对三角递推初始赋值
            wtemp=sin(0.5*theta);
            wpr = -2.0*wtemp*wtemp;
            wpi=sin(theta);
            wr=1.0;
            wi=0.0;
            for (i3=1;i3<=ifp1;i3+=ip1) {
                for (i1=i3;i1<=i3+ip1-2;i1+=2) {
                    for (i2=i1;i2<=ip3;i2+=ifp2) {
                        k1=i2;                 丹尼尔森—兰佐斯公式
                        k2=k1+ifp1;
                        tempr=(float)wr*data[k2]-(float)wi*data[k2+1];
                        tempi=(float)wr*data[k2+1]+(float)wi*data[k2];
                        data[k2]=data[k1]-tempr;
                        data[k2+1]=data[k1+1]-tempi;
```

```
                data[k1] += tempr;
                data[k1+1] += tempi;
            }
        }
        wr=(wtemp=wr)*wpr-wi*wpi+wr;    三角递推
        wi=wi*wpr+wtemp*wpi+wi;
    }
    ifp1=ifp2;
  }
  nprev *= n;
 }
}
```

## 12.5 二维和三维实数据的傅里叶变换

在图像处理领域中，二维 FFT 是特别重要。一幅图像通常表示为象素强度值的一个二维数组，并且是实数(通常为正数)。人们通常要求从图像中滤波出高频或低频的特殊分量，或者将图像与某些仪器的点扩展函数作卷积或解卷积，所以，FFT 的运用是十分有效的技术。

在三维情况下，FFT 的通常用法是，在代表三维空间方向的三维网格上为求位能(即电磁的或万有引力的)而求解泊松方程。那里源(物质分布或电荷分布)和所求位能也都是实数。在二维三维中具有大型的数组，通常存储量很受重视。所以，在尽可能的范围内，以"间址"来实 施 FFT 是很重要的。我们希望有一程序，其作用与多维 FFT 程序 **fourn** 相似(第12.4节)，但它是对实型输入数据操作而不复型数据操作。本节我们将给出这一程序。程序的导出与第12.3节中推导出的一维程序 **realft** 相似(希望在这一点上复习那节内容，特别是式(12.3.5))。

我们可以很方便地将式(12.4.3)中的独立变量 $n_1,\ldots,n_L$ 视为在波数空间的一个 $L$ 维向量，并且有整数格点上的值。而变换 $H(n_1,\ldots,n_L)$ 即可写成 $H(\vec{n})$。

很容易知道，变换 $H(n)$ 在它 $L$ 维的每维上是周期性的。特别是，若 $\vec{P}_1,\vec{P}_2,\vec{P}_3,\ldots$ 表示成向量 $(N_1,0,0,\ldots),(0,N_2,0,\ldots),(0,0,N_3,\ldots)$ 等等，则

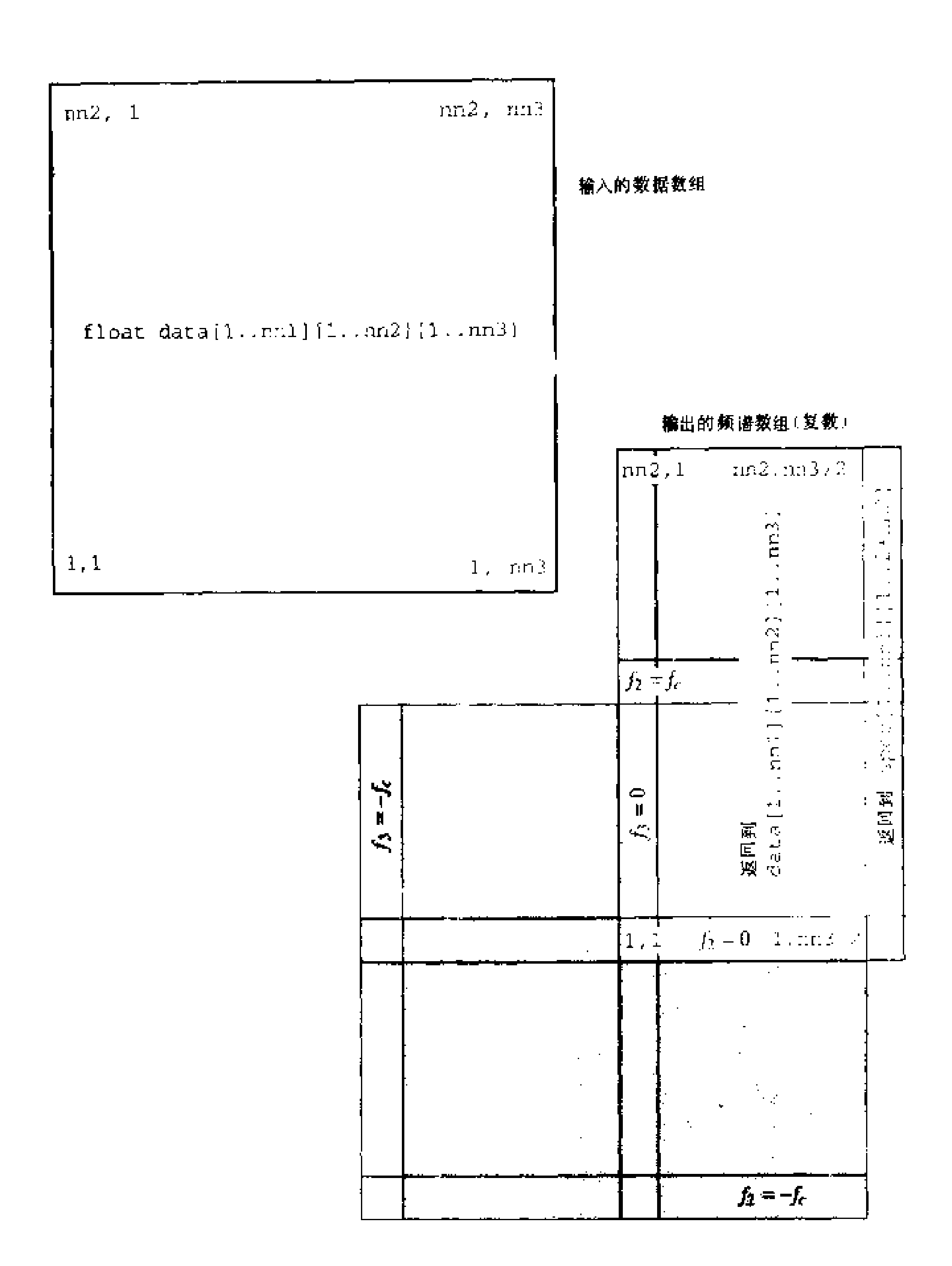
$$H(\vec{n}\pm\vec{P}_j)=H(\vec{n})\quad j=1,\ldots,L \tag{12.5.1}$$

对于任意输入数据——实型或复型，等式(12.5.1)都成立。当数据是实数，我们有附加对称性

$$H(-\vec{n})=H(\vec{n})^* \tag{12.5.2}$$

等式(12.5.1)和(12.5.2)表明，该全变换能够方便地从格点值 $n$ 的子集中获得，该子集有

$$\begin{aligned}
&0\le n_1\le N_1-1\\
&0\le n_2\le N_2-1\\
&\cdots\\
&0\le n_L<\frac{N_L}{2}
\end{aligned} \tag{12.5.3}$$




全部数组被假设为第一维(最左边)的范围[1..nn1],它伸出纸面。输入数组是一组三维实数组data[1..nn1][1..nn2][1..nn3](对于二维时可设为nn1=1)。输出数组可视为具有[1..nn1][1..nn2][1..nn3/2+1]维的一个复型数组,对应的频率$f_1$和$f_2$的分量以环绕次序存储,且只存储$f_3$的正频率部分(其余可通过对称性求得)。实际上,输出数据几乎都返回到输入数组data中,但部分存储在实数组speq[1..nn1][1..2*nn2]中,详见正文。

图12.5.1 程序rlft3中输入和输出数据的安排

事实上，这个子集的数值是过于完全了，因为在变换数值之间存在附加的对称关系，即在 $n_L=0$ 和 $n_L=N_L/2$ 处。但是，这些对称性是很复杂的，并且它们运用后会变得相当混乱。所以，我们仍在式(12.5.3)的网格子集上计算 FFT，尽管这样会要求少量额外的存储单元，即变换不是完全"同址"。(事实上，同址变换是可以实现的。但是我们发现，实际上，我们不可能向用户讲解清楚，如何分解出变换的输出，即如何寻找某些特殊频率处变换的实部和虚部分量)。

我们将实施三维情况 L＝3时的多维实傅里叶变换，并且有输入数据存储为一组实型三维数组 data[1..nn1][1..nn2][1..nn3]。这个实施允许用来处理二维数据，它同样有效而只需简单地选取 nn1＝1(注意，必须设第一维等于1)。至少逻辑上而言，输出谱将以复型三维数组成包地返回，其中我们能够调用 SPEC[1..nn1][1..nn2][1..nn3](参见式(12.5.3))。在变换的三维数据中，前两维的频率值分别为 $f_1$ 和 $f_2$，以环绕顺序存储，即零频率存在第一个序号值中，最小的正频率存储在第二个序号值中，最小复频率存储在最末序号中，等等(参见导出程序 **four1**和 **fourn** 的讨论)。三维数据中第三维仅仅返回频谱的正半部分。上页图12.5.1表明了这种逻辑存储结构。复输出谱的返回部分是图下方的阴影部分。

相对于逻辑而言，输出谱的物理结构要与逻辑结构少许不同，因为对于相同的实型和复型数组，C 语言没有合适的、易于移动的结构。下标范围 SPEC[1..nn1][1..nn2][1..nn3/2]返回到输入的数组 data[1..nn1][1..nn2][1..nn3]中，具有对应关系为

$$
\begin{aligned}
\mathrm{Re}(\mathrm{SPEC}[i1][i2][i3]) &= \mathrm{data}[i1][i2][2*i3-1] \\
\mathrm{Im}(\mathrm{SPEC}[i1][i2][i3]) &= \mathrm{data}[i1][i2][2*i3]
\end{aligned}
\tag{12.5.4}
$$

其余"同址"的值 SPEC[1..nn1][1..nn2][nn2/3＋1]是返回到二维浮点型数组 speq[1..nn1][1..2＊nn2]，其具有对应关系为

$$
\begin{aligned}
\mathrm{Re}(\mathrm{SPEC}[i1][i2][nn3/2+1]) &= \mathrm{speq}[i1][2*i2-1] \\
\mathrm{Im}(\mathrm{SPEC}[i1][i2][nn3/2+1]) &= \mathrm{speq}[i1][2*i2]
\end{aligned}
\tag{12.5.5}
$$

注意，speq 中包含频率量，其中第三个分量 $f_3$在奈斯特临界频率 $\pm f_c$ 上。事实上，在某些应用中，此值将被忽略或设成零，因为此处的值在正、负频率之间是被混迭的。

根据以上这些介绍，称之为 **rlft3**的实现程序有些虎头蛇尾。察看程序中最内层循环，它对最后面的变换下标实现方程式(12.3.5)。对 i3＝1 单独地编码是为了将数组 speq 填满以便替换输入数据的数组被覆盖。三个 for 所包围的循环(对下标 i2 、i3 和 i1 ，从内向外)事实上可以按任意顺序完成，因为它们的作用是可以互换的。我们选择了如程序中所示的的顺序有以下几点考虑：(i) i3 绝对不能是最内循环，因为若它是最内循环，则 wr 和 wi 的递推关系变得累赘了。(ii)在虚拟内存的计算机上，i1必须是外循环，因为数组 data 中的结果(以 C 语言的数组存储顺序)非常大，并且是以块序列的次序进行存取的。

请记住，与例行程序 **fourn** 中所完成的计算相关的复型 FFT 的实际工作相比较，程序 **rlft3**中所有的计算都可用一个对数因子被忽略。因为 C 语言中没有很方便的复型变量，所以复型数的操作完全是根据实部和虚部分别执行的。下面的例程 **rlft3**是基于赖比基(G. B. Rybicki)提出的较早的例程。

```
#include <math.h>
```

```
void rlft3(float ***data, float **speq, unsigned long nn1, unsigned long nn2,
    unsigned long nn3, int isign)
```

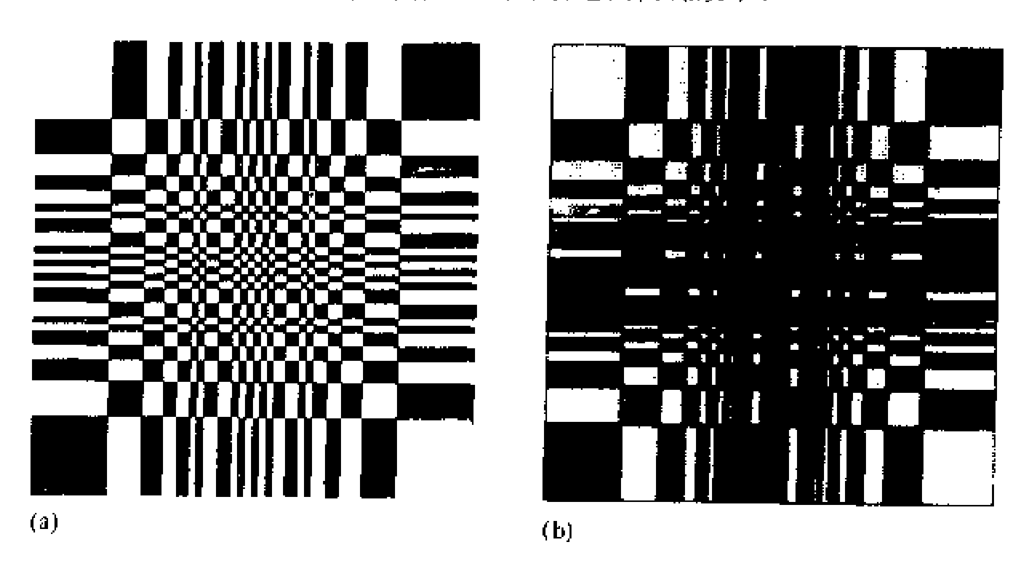
已知三维实型数组data[1..nn1][1..nn2][1..nn3](当nn1=1对应于二维数组的情况),该程序(对isign=1)返回两个复数组返回复FFT:输出数组data中包含第三个频率分量的零频率和正频率值,而speq[1..nn1][1..2*nn2]中包含第三频率分量的奈奎斯特临界频率值。第一(和第二)频率分量是以零频率、正频率和负频率的标准环绕次序存储的,详见正文中所述复数值如何安排的。对于isign=-1,执行逆变换(要乘上一个因子nn1*nn2*nn3/2),其输出数组data(看成实型数组)由输入数组data(看作复型)和speq得到,维数nn1,nn2,nn3通常必须是2的幂次。

```
{
    void fourn(float data[], unsigned long nn[], int ndim, int isign);
    void nrerror(char error_text[]);
    unsigned long i1,i2,i3,j1,j2,j3,nn[4],ii3;
    double theta,wi,wpi,wpr,wr,wtemp;
    float c1,c2,h1r,h1i,h2r,h2i;

    if (1+&data[nn1][nn2][nn3]-&data[1][1][1] != nn1*nn2*nn3)
        nrerror("rlft3: problem with dimensions or contiguity of data array\n");
    c1=0.5;
    c2 = -0.5*isign;
    theta=isign*(6.28318530717959/nn3);
    wtemp=sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi=sin(theta);
    nn[1]=nn1;
    nn[2]=nn2;
    nn[3]=nn3 >> 1;
    if (isign == 1) {                                正变换的情况
        fourn(&data[1][1][1]-1,nn,3,isign);          极大部分计算时间化在此处
        for (i1=1;i1<=nn1;i1++)
            for (i2=1,j2=0;i2<=nn2;i2++) {           周期地将data数组扩大成speq
                speq[i1][++j2]=data[i1][i2][1];
                speq[i1][++j2]=data[i1][i2][2];
            }
    }
    for (i1=1;i1<=nn1;i1++) {
        j1=(i1 != 1 ? nn1-i1+2 : 1);
        零频率是其自身的映射, 其他按照负频率以环绕次序定位
        in wrap-around order.
        wr=1.0;                                        三角递推初始赋值
        wi=0.0;
        for (ii3=1,i3=1;i3<=(nn3>>2)+1;i3++,ii3+=2) {
            for (i2=1;i2<=nn2;i2++) {
                if (i3 == 1) {                           等式 (12.3.5).
                    j2=(i2 != 1 ? ((nn2-i2)<<1)+3 : 1);
                    h1r=c1*(data[i1][i2][1]+speq[j1][j2]);
                    h1i=c1*(data[i1][i2][2]-speq[j1][j2+1]);
                    h2i=c2*(data[i1][i2][1]-speq[j1][j2]);
                    h2r= -c2*(data[i1][i2][2]+speq[j1][j2+1]);
                    data[i1][i2][1]=h1r+h2r;
                    data[i1][i2][2]=h1i+h2i;
                    speq[j1][j2]=h1r-h2r;
                    speq[j1][j2+1]=h2i-h1i;
                } else {
                    j2=(i2 != 1 ? nn2-i2+2 : 1);
                    j3=nn3+3-(i3<<1);
                    h1r=c1*(data[i1][i2][ii3]+data[j1][j2][j3]);
                    h1i=c1*(data[i1][i2][ii3+1]-data[j1][j2][j3+1]);
                    h2i=c2*(data[i1][i2][ii3]-data[j1][j2][j3]);
                    h2r= -c2*(data[i1][i2][ii3+1]+data[j1][j2][j3+1]);
                    data[i1][i2][ii3]=h1r+wr*h2r-wi*h2i;
                    data[i1][i2][ii3+1]=h1i+wr*h2i+wi*h2r;
                    data[j1][j2][j3]=h1r-wr*h2r+wi*h2i;
                    data[j1][j2][j3+1]= -h1i+wr*h2i+wi*h2r;
```

```
            }
        }
        wr=(wtemp=wr)*wpr-wi*wpi+wr;              实现递推
        wi=wi*wpr+wtemp*wpi+wi;
    }
  }
  if (isign == -1)                                 逆变换的情况
      fourn(&data[1][1][1]-1,nn,3,isign);
}
```

现在，我们给出几个程序段，说明如何对二维和三维数据调用 **rlft3** 例程。注意，这个例程实际上并不能区分二维和三维数据：二维也和三维一样处理，只是第一维数组的长度设为1。因为第一维是在最外层循环，所以实际上没有引起任何低效率。



(a)一幅灰度只有纯黑或纯白的二维图像。(b)(a)图像用程序 **rlft3** 经过低通滤波后的图像，具有明显标尺特性的区域变成了灰色。

图12.5.2

第一个程序段是二维数据的 FFT，允许对它作某种处理，如滤波处理，然后取其逆变换。图12.5.2显示了这段代码使用的例子：当一幅轮廓鲜明的图像的高频空间分量被因子(此处)$\max(1-6f^2/f_c,0)$抑制后，轮廓变得模糊起来。第二个程序段举例说明一个三维 FFT，其中三维方向具有不同的长度。第三个程序是一个卷积的例子，它可以用来计算三维分布信源所产生的位能。

```
#include <stdlib.h>
#include "nrutil.h"
#define N2 256                              注意第一个分量必须设为1
#define N3 256

int main(void) /* example1 */
  这段程序说明如何对一幅256×256的数字图像进行滤波.
{
    void rlft3(float ***data, float **speq, unsigned long nn1,
        unsigned long nn2, unsigned long nn3, int isign);
    float ***data, **speq;

    data=f3tensor(1,1,1,N2,1,N3);
```

```
    speq=matrix(1,1,1,2*N2);
/*  ...*/                             这里图像必须装入 data 数组
    rlft3(data,speq,1,N2,N3,1);
/*  ...*/                             这里数组 data 和 speq 必须乘以适当的(频率的)滤波因子
    rlft3(data,speq,1,N2,N3,-1);
/*  ...*/                             这里已被滤波的图象应该从 data 数组中卸下
    free_matrix(speq,1,1,1,2*N2);
    free_f3tensor(data,1,1,1,N2,1,N3);
    return 0;
}

#define N1 32
#define N2 64
#define N3 16

int main(void)          /* example2 */
  这段程序说明如何对一个32×64×16三维实型数组作 FFT。
{
    void rlft3(float ***data, float **speq, unsigned long nn1,
        unsigned long nn2, unsigned long nn3, int isign);
    int j;
    float ***data, **speq;

    data=f3tensor(1,N1,1,N2,1,N3);
    speq=matrix(1,N1,1,2*N2);
/*  ...*/                             这里装载 data 数组
    rlft3(data,speq,N1,N2,N3,1);
    /*  ...*/                         这里卸下 data 和 speq
    free_matrix(speq,1,N1,1,2*N2);
    free_f3tensor(data,1,N1,1,N2,1,N3);
    return 0;
}

#define N 32

int main(void)      /* example3 */
  这段程序说明如何对两个32×32×32实型三维数组实行卷积，结果替换了原第一个数组。
{
    void rlft3(float ***data, float **speq, unsigned long nn1,
        unsigned long nn2, unsigned long nn3, int isign);
    int j;
    float fac,r,i,***data1,***data2,**speq1,**speq2,*sp1,*sp2;

    data1=f3tensor(1,N,1,N,1,N);
    data2=f3tensor(1,N,1,N,1,N);
    speq1=matrix(1,N,1,2*N);
    speq2=matrix(1,N,1,2*N);

 /*    ...*/
    rlft3(data1,speq1,N,N,N,1);           二个输入数组的FFT
    rlft3(data2,speq2,N,N,N,1);
    fac=2.0/(N*N*N);                      为了得到标准逆变换所需的因子
    sp1 = &data1[1][1][1];
    sp2 = &data2[1][1][1];
    for (j=1;j<=N*N*N/2;j++) {            注意：怎样使用指针sp1和sp2组成单个for循环而
        r = sp1[0]*sp2[0] - sp1[1]*sp2[1];       替换三个嵌套的for的循环
        i = sp1[0]*sp2[1] + sp1[1]*sp2[0];
        sp1[0] = fac*r;
        sp1[1] = fac*i;
        sp1 += 2;
        sp2 += 2;
```

```
    }
    sp1 = &speq1[1][1];
    sp2 = &speq2[1][1];
    for (j=1;j<=N*N;j++) {
        r = sp1[0]*sp2[0] - sp1[1]*sp2[1];
        i = sp1[0]*sp2[1] + sp1[1]*sp2[0];
        sp1[0] = fac*r;
        sp1[1] = fac*i;
        sp1 += 2;
        sp2 += 2;
    }
    rlft3(data1,speq1,N,N,N,-1);          个FFT乘积的逆FFT
/*    ...*/
    free_matrix(speq2,1,N,1,2*N);
    free_matrix(speq1,1,N,1,2*N);
    free_f3tensor(data2,1,N,1,N,1,N);
    free_f3tensor(data1,1,N,1,N,1,N);
    return 0;
}
```

为了将程序 **rlft3** 扩展成四维，读者可以简单地增加一个附加的(最外层)、对应段位的 for 循环，类似于原先的 i1。(也可将这程序修改成任意维，如同程序 **fourn**。这对读者是一个很好的编程练习。)

## 12.6 外部存储和局部内存的 FFT

有时在工作中，需要计算非常大的数据集的傅里叶变换，它大于计算机物理内存的大小。在这种情况下，数据必须存在某些外部的介质中，如磁盘或光盘。因此，需要一种算法，它能导出某些连续地通过外部数据的管理数字，这些数字能连续读取。

事实上，FFT 发明不久，辛莱托(Singleton)[1]就已提出了这种算法。这种算法要求有四个顺序的存储驱动器，每个驱动器的容量是输入数据的一半。通常，输入数据前一半在一个驱动器内，另一半在另一个驱动器内。

辛莱托算法是基于通过下述的操作步骤，$2^M$ 个值的位序颠倒是可能的：第一步，从二个输入驱动器交替地读入数值，并且写入到一个输出驱动器中(直到它存满数据的一半)，然后写入到另一个输出驱动器。第二步，将输出驱动器变成输入驱动器，并将原输入驱动器变成输出驱动器。现在，从第一个驱动器复制两个数值，然后从第二个驱动器中复制两个数值，把它们写入(如前)到第一个输出驱动器，写满后就写入第二个输出驱动器。依次进行，同时读 4，8…等输入数据的操作过程。直到完成了 $M-1$ 次操作后，数据便按位序颠倒次序排列。

其次，根据辛莱托算法，本质上位序颠倒技术的过程和实现丹尼尔森-兰佐斯组合公式(12.2.3)的每步过程可以交替地进行，其过程大致如下：起始和前述相同，在每一驱动器内各读入一半的输入数据。第一步，从每个输入驱动器中读入一个复型数值，形成两个组合，并且将各自写入两个输出驱动器之一。在这些"计算"过程以后，这些驱动器被改写，并且实现了"置换"过程，其中一组数值从第一个输入驱动器读入，又交换地被写入第一个和第二个输出驱动器；当第一个输入驱动器被耗尽，第二个输入驱动器同样地处理。这种计算和置换过程重复 $M-K-1$ 次，此处 $2^K$ 是程序有效的内部缓冲器。第二个相位的计算由最后 $K$ 次计算过程组成。现在，第二个相位与第一个相位的区别是在计算过程中，由于置换是局部的，足以可能实现，所以在第二个相位中不存在分离的置换步骤。处理数据总共需 $2M-K-2$ 步。

以下是辛莱托算法的实现程序，它基于[1]。

```
#include <stdio.h>
#include <math.h>
#include "nrutil.h"
#define KBF 128
```

void fourfs(FILE *file[5], unsigned long nn[], int ndim, int isign)

存储在外部介质的大数组的一维或多维傅里叶变换，对于输入，ndim 是维数的个数，和 nn[1..ndim] 表示每维的长度（实部和虚部数值对的个数），它们必须都是2的幂次。file[1..4]是指向临时文件流式指针，每个大小要足以放下一半数据。这四个流式指针在系统的"二进制"模式（相对为"文本"模式）下必须是开放的。输入数据按 C 语言标准次序排列，以自然浮点数形式将数据的前一半存入 file[1]，其后一半存入 file[2]。每次缓冲读或写所处理的数据个数为 KBF。isign 为1是 FFT，为−1是 FFT 的逆变换。对于输出，数组 file 中的数值可能被替换：结果的前一半存入 file[3]，结果的后一半存入 file[4]。注意：当 ndim>1，输出是以列形式存储的，即不按 C 语言的标准形式的顺序；换言之，现在的输出是程序 **fourn** 产生的输出结果的转置。

```
{
    void fourew(FILE *file[5], int *na, int *nb, int *nc, int *nd);
    unsigned long j,j12,jk,k,kk,n=1,mm,kc=0,kd,ks,kr,nr,ns,nv;
    int cc,na,nb,nc,nd;
    float tempr,tempi,*afa,*afb,*afc;
    double wr,wi,wpr,wpi,wtemp,theta;
    static int mate[5] = {0,2,1,4,3};

    afa=vector(1,KBF);
    afb=vector(1,KBF);
    afc=vector(1,KBF);
    for (j=1;j<=ndim;j++) {
        n *= nn[j];
        if (nn[j] <= 1) nrerror("invalid float or wrong ndim in fourfs");
    }
    nv=1;
    jk=nn[nv];
    mm=n;
    ns=n/KBF;
    nr=ns >> 1;
    kd=KBF >> 1;
    ks=n;
    fourew(file,&na,&nb,&nc,&nd);
                                            此处开始第一个相位的变换
    for (;;) {                              计算过程开始
        theta=isign*3.141592653589793/(n/mm);
        wtemp=sin(0.5*theta);
        wpr = -2.0*wtemp*wtemp;
        wpi=sin(theta);
        wr=1.0;
        wi=0.0;
        mm >>= 1;
        for (j12=1;j12<=2;j12++) {
            kr=0;
            do {
                cc=fread(&afa[1],sizeof(float),KBF,file[na]);
                if (cc != KBF) nrerror("read error in fourfs");
                cc=fread(&afb[1],sizeof(float),KBF,file[nb]);
                if (cc != KBF) nrerror("read error in fourfs");
                for (j=1;j<=KBF;j+=2) {
                    tempr=((float)wr)*afb[j]-((float)wi)*afb[j+1];
                    tempi=((float)wi)*afb[j]+((float)wr)*afb[j+1];
                    afb[j]=afa[j]-tempr;
                    afa[j] += tempr;
                    afb[j+1]=afa[j+1]-tempi;
                    afa[j+1] += tempi;
                }
                kc += kd;
                if (kc == mm) {
                    kc=0;
                    wr=(wtemp=wr)*wpr-wi*wpi+wr;
                    wi=wi*wpr+wtemp*wpi+wi;
                }
```

```
            cc=fwrite(&afa[1],sizeof(float),KBF,file[nc]);
            if (cc != KBF) nrerror("write error in fourfs");
            cc=fwrite(&afb[1],sizeof(float),KBF,file[nd]);
            if (cc != KBF) nrerror("write error in fourfs");
        } while (++kr < nr);
        if (j12 == 1 && ks != n && ks == KBF) {
            na=mate[na];
            nb=na;
        }
        if (nr == 0) break;
    }
    fourew(file,&na,&nb,&nc,&nd);          置换过程开始
    jk >>= 1;
    while (jk == 1) {
        mm=n;
        jk=nn[++nv];
    }
    ks >>= 1;
    if (ks > KBF) {
        for (j12=1;j12<=2;j12++) {
            for (kr=1;kr<=ns;kr+=ks/KBF) {
                for (k=1;k<=ks;k+=KBF) {
                    cc=fread(&afa[1],sizeof(float),KBF,file[na]);
                    if (cc != KBF) nrerror("read error in fourfs");
                    cc=fwrite(&afa[1],sizeof(float),KBF,file[nc]);
                    if (cc != KBF) nrerror("write error in fourfs");
                }
                nc=mate[nc];
            }
            na=mate[na];
        }
        fourew(file,&na,&nb,&nc,&nd);
    } else if (ks == KBF) nb=na;
    else break;
}
j=1;
```

此处开始第二个相位的变换。现在，余留下的置换足以同址完成

```
for (;;) {
    theta=isign*3.141592653589793/(n/mm);
    wtemp=sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi=sin(theta);
    wr=1.0;
    wi=0.0;
    mm >>= 1;
    ks=kd;
    kd >>= 1;
    for (j12=1;j12<=2;j12++) {
        for (kr=1;kr<=ns;kr++) {
            cc=fread(&afc[1],sizeof(float),KBF,file[na]);
            if (cc != KBF) nrerror("read error in fourfs");
            kk=1;
            k=ks+1;
            for (;;) {
                tempr=((float)wr)*afc[kk+ks]-((float)wi)*afc[kk+ks+1];
                tempi=((float)wi)*afc[kk+ks]+((float)wr)*afc[kk+ks+1];
                afa[j]=afc[kk]+tempr;
                afb[j]=afc[kk]-tempr;
                afa[++j]=afc[++kk]+tempi;
                afb[j++]=afc[kk++]-tempi;
                if (kk < k) continue;
                kc += kd;
                if (kc == mm) {
                    kc=0;
                    wr=(wtemp=wr)*wpr-wi*wpi+wr;
```

```
                wi=wi*wpr+wtemp*wpi+wi;
            }
            kk += ks;
            if (kk > KBF) break;
            else k=kk+ks;
        }
        if (j > KBF) {
            cc=fwrite(&afa[1],sizeof(float),KBF,file[nc]);
            if (cc != KBF) nrerror("write error in fourfs");
            cc=fwrite(&afb[1],sizeof(float),KBF,file[nd]);
            if (cc != KBF) nrerror("write error in fourfs");
            j=1;
        }
        }
        na=mate[na];
    }
    fourew(file,&na,&nb,&nc,&nd);
    jk >>= 1;
    if (jk > 1) continue;
    mm=n;
    do {
        if (nv < ndim) jk=nn[++nv];
        else {
            free_vector(afc,1,KBF);
            free_vector(afb,1,KBF);
            free_vector(afa,1,KBF);
            return;
        }
    } while (jk == 1);
  }
}
```

```
#include <stdio.h>
#define SWAP(a,b) ftemp=(a);(a)=(b);(b)=ftemp

void fourew(FILE *file[5], int *na, int *nb, int *nc, int *nd)
    用于 fourfs 的实用程序。重环绕和重记数四个文件
{
    int i;
    FILE *ftemp;

    for (i=1;i<=4;i++) rewind(file[i]);
    SWAP(file[2],file[4]);
    SWAP(file[1],file[3]);
    *na=3;
    *nb=4;
    *nc=1;
    *nd=2;
}
```

对于一维数据,辛莱托的算法处理输出结果,基本上和标准的FFT(即**four1**程序)有相同的排列顺序。而对于多维数据,算法输出是常规排列(即程序**fourn**的输出)的转置。一般地,这种独特性是这个算法固有的特件,只是不太方便。对于卷积,计算非标准排列的两变换的部分对部分乘积,然后对结果计算逆变换,这就有些不方便。注意,若不同维数的长度不全相同,则必须在实行逆变换之前,颠倒nn[1..ndim]中的数值次序(因此给出转置的维数)。也必须注意,和程序**fourn**相同,实现变换并且然后实现逆变换,必须逆变换的结果乘上原数据所有维数的长度之积。

我们留下一个习题给读者,请读者将程序**fourfs**的输出重新排成标准序形式,需通过外部存储数据的过程。但我们怀疑这种重排是否真正必须的。

有时，读者需要修改 **fourfs** 以适合自己特殊应用。例如，程序中所写 $KBF = 2^K$，它起了双重角色：它是内部缓冲器的大小，也是无格式化读和写记录大小。后者由机器的 I/O 设备所允许的大小而受限。修改的简单方法是对于大的 KBF 数，采用双倍读取，就可减少一些过程的次数。

修改 **fourfs** 的另一情况是为了用惯例的算法完成高效率的 FFT（惯例算法中，内存引用是极其非本地的），这时虚拟内存有足够的地址空间，而没有足够的物理内存。在这种情况下，必须用将数组 afa、afb 和 afc 映射到地址空间的方法来代替读、写和重绕。换言之，这些数组被引用单个数据的数组而替代，随着偏移而修改了 **fourfs** 程序中实现 I/O 的操作。这种算法其内存引用的局部块是 KBF 的大小。尽管它要求花费二倍于同址 FFT 所需的虚拟内存，但有时其执行速度却大大地提高。

### 参考文献和进一步读物：


Singleton, R. C. 1967, *IEEE Transactions on Audio and Electroacoustics*, vol. AU-15, PP. 91～97. [1]

Oppenheim, A. V, and Schafer, R. W. 1989, *Discrete-Time Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 9.

# 第十三章　傅里叶和谱的应用

## 13.0　引言

傅里叶方法已经使科学和工程领域，从射电天文到医学映象，从地震学到光谱学，都发生了重大的变革。本章，我们讨论一些傅里叶和谱方法的基本应用，它们使这些变革成为可能。

称"傅里叶"为数值的，其反应就应该是"FFT"，如同巴甫洛夫(Pavlovian)的条件反射。的确，傅里叶方法的广泛应用原则上应该归功于快速傅里叶变换的存在。暂且离开主题来说：假如有人将任意一个非平凡的算法加快一个百万因子或更多，则在世界上将开辟一条寻找这种算法有效应用的途径。FFT 的最直接的应用就是数据的卷积和解卷积(第13.1节)、相关和自相关(第13.2节)、最佳滤波(第13.3节)、功率谱估计(第13.4节)以及傅里叶积分的计算(第13.9节)。

然而，重要的是 FFT 方法，并非是谱分析的全部内容。第13.5节将简要介绍时域数字滤波器的领域。在谱域中，FFT 的一个限制因素是，常常将函数的傅里叶变换表示成 $z$ 的多项式，即 $z=\exp(2\pi if\Delta)$(见式(12.1.7))。有时候，过程的谱形状不能很好地由这种形式来表示。而另外一种形式，它允许谱在 $z$ 处有极点存在，并且常应用于线性预测技术(第13.6节)和最大熵谱估计(第13.7节)。

所有 FFT 方法的另一个重要的限制因素是，要求输入数据在均匀间隔内取样。对于非规则的或者非完整的取样数据，其它方法(尽管速度慢)是有效的，将在第13.8节中讨论。

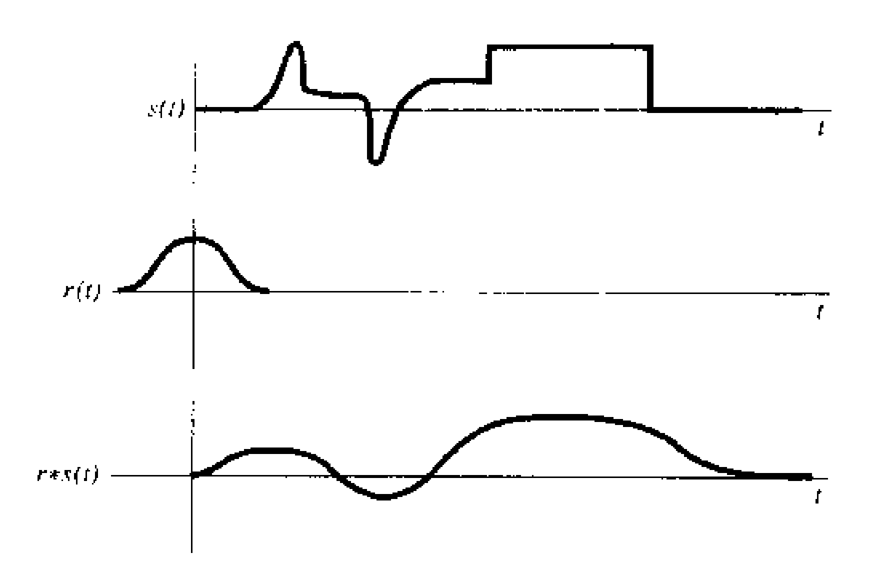
所谓小波方法就是函数的空间表示，它既不是在时间域，也不是在谱域，而是有些在这两者之间。第13.10节将介绍这方面内容。最后，第13.11节有点离题，将讨论傅里叶取样定理的数值应用。

## 13.1　使用 FFT 作卷积和解卷积

我们已经在式(12.0.8)中对连续情况定义了两个函数的**卷积**，而且得出如式(12.0.9)的**卷积定理**。这个定理表明，两个函数卷积的傅里叶变换等于它们各自傅里叶变换的乘积。现在，我们来处理离散的情况。我们首先提及卷积是一个有用的过程，然后讨论如何运用 FFT 有效地计算卷积。

两个函数 $r(t)$ 和 $s(t)$ 的卷积，记作 $r*s$，从数学上来说，它等于这两函数反序的卷积 $s*r$。然而，在大多数应用中，这两个函数有十分不同的含义和特性。其中 $s$ 函数，是一个典型信号或数据流，在时间上无期限地进行下去(或者说，无论怎样它可以是一个适当的独立变量)。而另一函数 $r$ 是一个"响应函数"，它是从最大值的两边都下降到零的有峰函数。卷积的结果是按照响应函数 $r(t)$ 提供的方法，在时间上涂抹去信号 $s(t)$，如图13.1.1所示。特别是，

我们假设在某一时刻 $t_0$ 出现的单位面积的峰值函数或 $\delta$-函数 $s$，其卷积结果将被涂抹成响应函数本身形状，只是时间从0移到 $t_0$，如同 $r(t-t_0)$。




信号 $s(t)$ 与响应函数 $r(t)$ 卷积。因为响应函数比原来信号的某些特征更为明显，因此在卷积中这些特征要被"洗掉"。在无附加噪声的情况下，这个过程可以被解卷积逆转过来。

图13.1.1　两函数的卷积示例

在离散情况下，信号 $s(t)$ 被它的相等时间间隔上的样本值 $s_j$ 所取代。响应函数同样也是离散数集 $r_k$，并且有以下解释：$r_0$ 说明了将某一个通道中($j$ 的某一特定值)的多少重输入信号被复制到等同的输出通道(同样的 $j$ 值)；$r_1$ 说明在第 $j$ 通道中有多少重输入信号又被复制到第 $j+1$ 通道；$r_{-1}$ 说明有多少重信号被复制到第 $j-1$ 通道；对 $r_k$ 的下标 $k$ 取正负值即可以此类推，图13.1.2阐明了这种情况。

例：一个 $r_0=1$，其它 $r_k$ 都等于零的响应函数，它正是一个恒等的滤波器：一个信号与此响应函数的卷积给出同样的此信号。另一个例子是响应函数 $r_{14}=1.5$，而其余 $r_k$ 都等于零，这产生了卷积输出，它是输入信号乘上 1.5 并且延迟 14 个取样时间间隔。

显而易见，我们已经用文字描述了如下的定义，它是具有有限持续时间 $M$ 的响应函数之离散卷积：

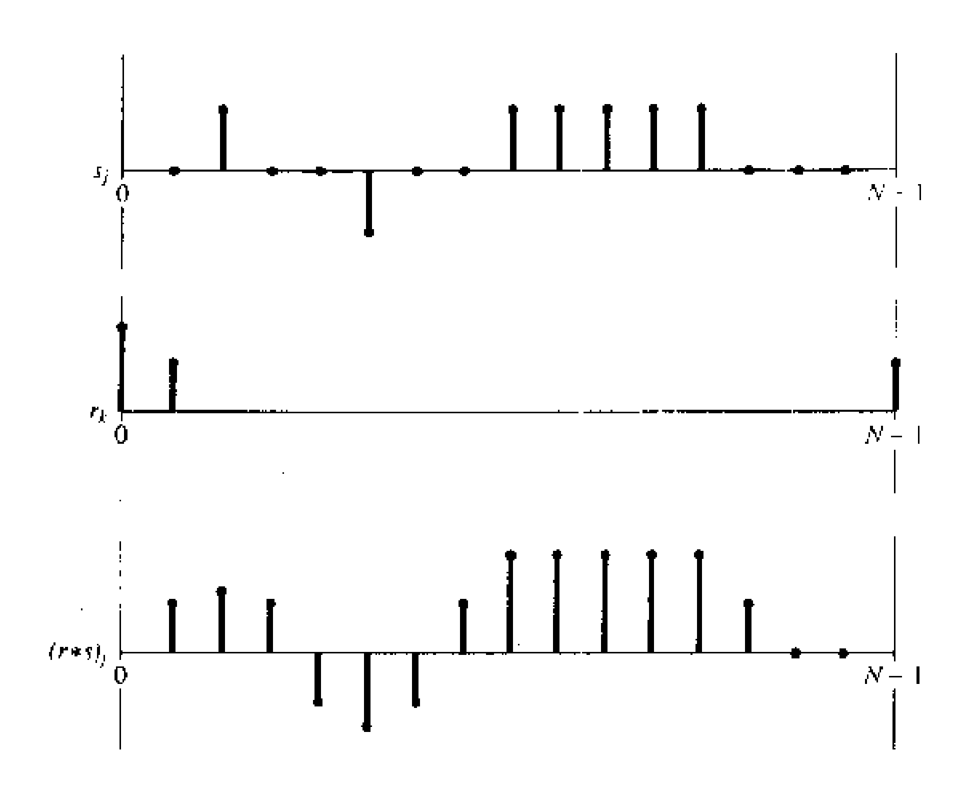
$$(r * s)_j \equiv \sum_{k=-M/2+1}^{M/2} s_{j-k} r_k \qquad (13.1.1)$$

如果一个离散响应函数只在某范围 $-M/2<k\leqslant M/2$ 内为非零，其中 $M$ 是足够大的偶整数，那么响应函数称为**有限脉冲响应**($FIR$)，其**持续时间**为 $M$(注意，我们定义 $M$ 为 $r_k$ 的非零值的个数；这些值跨越了 $M-1$ 个取样时间间隔)。在多数实践中，$M$ 有限的情况是有意义的情况，这是因为响应确实在一个有限的持续时间，或者因为我们选择在某点将它截断，并用一个有限持续时间来逼近它。

**离散卷积定理**：如果信号 $s_j$ 是**周期的**，其周期为 $N$，并且它完全由 $N$ 个值 $s_0,\cdots,s_{N-1}$ 所确定，则它与一个有有限持续时间 $N$ 之响应函数的卷积，就是离散傅里叶变换对之一，

$$\sum_{k=-N/2+1}^{N/2} s_{j-k} r_k \iff S_n R_n \qquad (13.1.2)$$

其中 $S_n(n=0,\ldots,N-1)$ 是数值 $s_j(j=0,\ldots,N-1)$ 的离散傅里叶变换，而 $R_n(n=0,\ldots,N-1)$ 是数值 $r_k(k=0,\ldots,N-1)$ 的离散傅里叶变换。$r_k$ 的这些值与范围 $k=-N/2+1,\ldots,N/2$ 内的值相等，但按环绕顺序，如第12.2节末尾所描述。



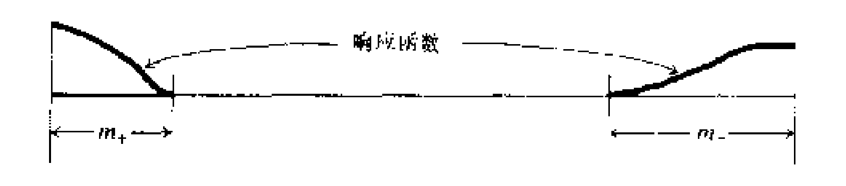

注意，对于负时间响应函数如何被环绕，并且被存储在 $r_k$ 数组最右端。

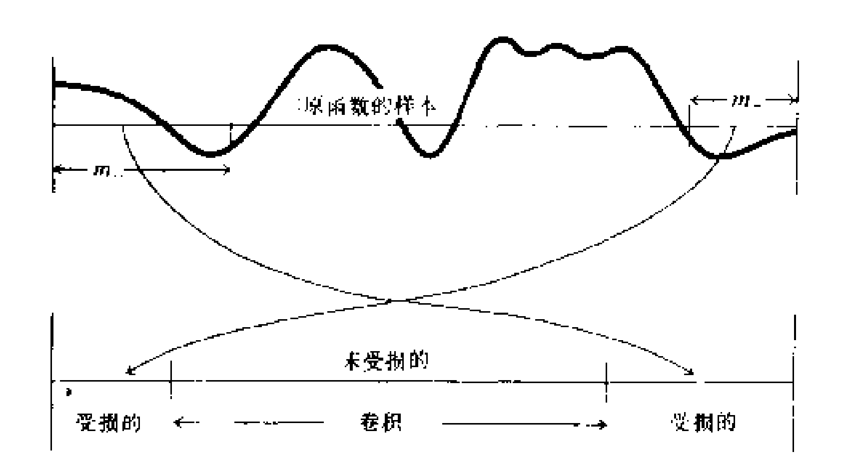
图13.1.2　离散取样函数的卷积

## 13.1.1　用零元填充的终端效应处理

离散卷积定理假定了两个非通用的事实。首先，它假定输入信号是周期的，但实际数据要么是不再重复地永远取下去，要么是由一个有限长度的非周期信号伸展所组成。第二，卷积定理将响应函数的持续时间取得与数据的周期相同，两者都是 $N$。我们需要先围绕这两种约束进行工作。

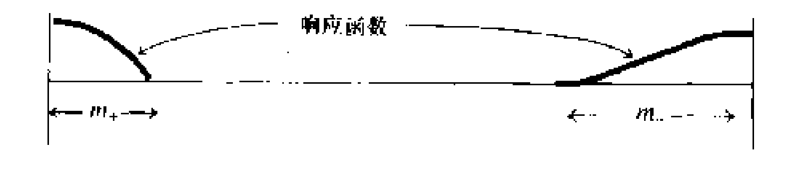
第二种约束是非常简单明了的。人们几乎永远感兴趣于那种持续时间 $M$ 远远小于数据集长度 $N$ 的响应函数。在这种情况下，人们简单地通过零元填充来扩展响应函数的长度到 $N$，即在 $M/2\leqslant k\leqslant N/2$ 和 $-N/2+1\leqslant k\leqslant -M/2+1$ 内，定义 $r_k=0$。而处理第一种约束，则需要更多要求。因为卷积定理轻率地假定数据是周期性的，这可能造成数据流 $s_{N-1}$，$s_{N-2}$，等远端的环绕数据对第一输出通道 $(r*s)_0$ 不真实地"污染"(如图13.1.3所示)。因此，我们必须在 $s_j$ 向量的端部建立一个零元填充值的缓冲区，为的是使得这种污染为零。在这缓冲区中，我们需要多少个零值呢？确切地说，应与响应函数非零的的最小负索引一样多。例如，如果 $r_{-3}$ 非零，而 $r_{-4}$，$r_{-5}\ldots$ 均为零，则我们需要在数据端部填充三个零元：$s_{N-3}=s_{N-2}=s_{N-1}=0$。这些零将保护第一输出通道 $(r*s)_0$ 不受环绕污染。显然，第二输出通道和随后的通道也都受这些零元的保护。设 $k$ 表示填充零元的个数，则最后实际输入数据点的个数为 $s_{N-K-1}$。

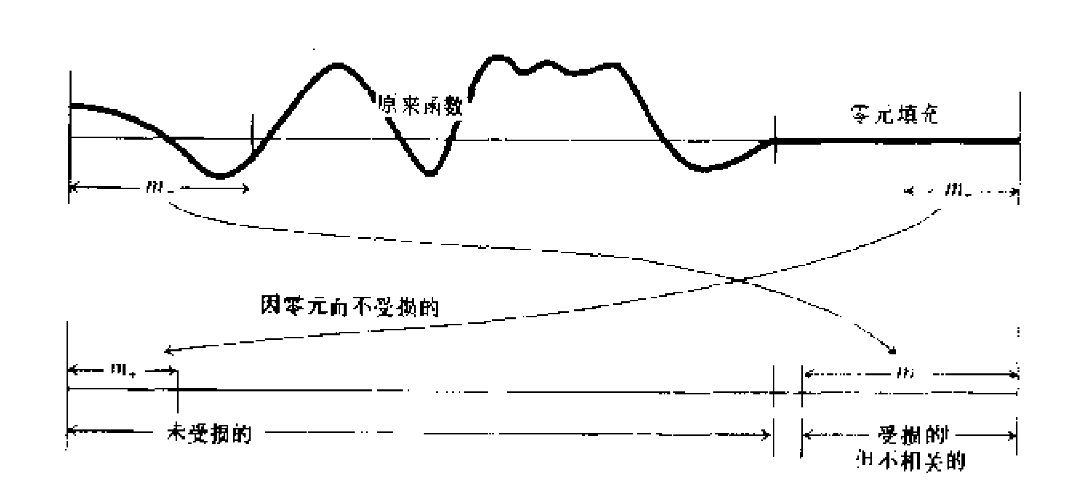



不仅响应函数的环绕必须看成是循环的，而且原样本函数也必须如此。所以，原函数每一端的一部分都被它与响应函数的卷积错误地环绕。

图13.1.3　有限段函数卷积中的环绕问题




原来函数被零元进行了扩展，它服务于二个目的：当零元环绕回来，它们并没有干扰真正的卷积；并且当原来函数环绕到零域时，这个区域可以丢弃。

图13.1.4　解决环绕问题的零元填充。

那么，现在关于这个最后输出通道的污染情况如何呢？由于现在数据以 $s_{N-1}$ 结尾，故有意义的最后输出通道是$(r*s)_{N-1}$。这一通道可能被输入通道 $s_0$的环绕而污染，除非数字 $K$ 也足够大，足以照顾到最大正下标 $k$，而其响应函数 $r_k$ 是非零的。例如，如果 $r_1$ 直到 $r_6$ 都不为零，而 $r_7, r_8, \ldots$ 都为零，则我们在数据末端至少需要 $k=6$ 个填充零元：$s_{N-6}=\ldots=s_{N-1}=0$。

总而言之，我们在数据一端用大量零元来填充，填充个数等于响应函数的最大正的持续时间或者最大负的持续时间，决定于哪一段更大一些。（对于持续时间为 $M$ 的对称响应函数，则只需 $M/2$个零元填充）。将这一操作和上述响应函数 $r_k$ 的填充相结合，我们便可有效地形成不期望的人工因素的周期性数据。上页图13.1.4阐述了这个问题。

### 13.1.2 FFT 对卷积的使用

用零元填充而完善的数据，现在是一组实数 $s_j(j=0,\ldots,N-1)$以及响应函数被零元填充成持续时间为 $N$，并且以环绕顺序排列。（一般说来，这意味着在数组的中部，$r_k$ 的一大段邻近取零值，而非零值密集在数组的两个端部。）现在就可按以下步骤计算离散卷积：用 FFT 算法计算 $s$ 和 $r$ 的离散傅里叶变换，按逐个分量将两个变换乘在一起，记住这两个变换是由复数组成的。然后用 FFT 算法计算乘积的离散傅里叶逆变换。其结果就是卷积 $r*s$。

什么是**解卷积**呢？解卷积就是取消已被涂改数据集的过程，该涂改是在一个已知函数的影响下发生的，例如，由于不太完善的测量装置的已知影响。解卷积的定义和卷积式(13.1.1)是相同的，只是现在左端是已知的，并且式(13.1.1)可以被认为是未知量 $s_j$ 的 $N$ 个线性方程的方程组，在式(13.1.1)的时域中，求解这些联立线性方程，在大多数情况下是不现实的，但是 FFT 使问题几乎变成了平常而轻松。我们不再把信号的变换与响应函数的变换相乘以得到卷积的变换，而是用响应的变换去除已知卷积的变换，以得到解卷积信号的变换。

从数学上看，如果响应函数变换在某值 $R_n$ 处恰好为零，因此我们不能用它做除数，则这个过程可能要出错。这说明原来的卷积确实丧失了在那个频率上的所有信息，所以那个频率分量的重建是不可能的。然而，除数学问题以外，解卷积过程还有其它实践中的缺点，这过程一般对输入数据中的噪声非常敏感，并且对于已知响应函数 $r_k$ 的精确度也很敏感。由于这些原因，即使对解卷积是完全合理的尝试，有时也可能会产生荒谬的结果。在这种情况下，可能要利用附加的**最佳滤波**过程，这将于第13.3节中讨论。

下面是关于卷积和解卷积的程序，其采用第12.2节中程序 **four1**实施 FFT。因为数据和响应都是实型而非复数型的，所以两者变换可用第12.3节中提到的技巧同时进行。这样，该程序恰好在计算 FFT 时调用一次，而计算逆 FFT 时也调用一次。数据假定存储在一个浮点型数组 data[1..n]中，其中 $n$ 是 2 的整数幂。响应函数假定以环绕次序存储在数组 respns[1..n]的一个子数组 respns[1..m]中，$m$ 值可以是小于或等于 $n$ 的任意奇整数。因为程序要做的第一件事，就是将响应函数重新复制成 respn[1..n]中适当的环绕顺序。结果返回到 ans 中。

```
#include "nrutil.h"

void convlv(float data[], unsigned long n, float respns[], unsigned long m,
    int isign, float ans[])
```

实型数据集 data[1..n]与响应函数 respns[1..n]的卷积和解卷积.(包括任何为用户提供的零元填充),响应函数必须以环绕顺序存储在数组 respns 的前 m 个元素中,其中 m 为≤n 的奇整数,环绕顺序意味着数组 respns 的前一半包含正时间上的脉冲响应函数,而数组的后一半包含负时间上的脉冲响应函数,计算从最高元素 respns[m]开始而向下,在输入时,isign 是+1作卷积,isign 是-1作解卷积。结果返回到 ans 的前 n 个分量。为了与 **twofft** 程序兼容,ans 必须提供程序调用中所需的维数[1..2*n],n 必须是2的整幂次

```
{
    void realft(float data[], unsigned long n, int isign);
    void twofft(float data1[], float data2[], float fft1[], float fft2[],
        unsigned long n);
    unsigned long i,no2;
    float dum,mag2,*fft;

    fft=vector(1,n<<1);
    for (i=1;i<=(m-1)/2;i++)                  把 respns 取成 n 长数组
        respns[n+1-i]=respns[m+1-i];
    for (i=(m+3)/2;i<=n-(m-1)/2;i++)          填充零元
        respns[i]=0.0;
    twofft(data,respns,fft,ans,n);            立刻作两个函数的FFT
    no2=n>>1;
    for (i=2;i<=n+2;i+=2) {
        if (isign == 1) {
            ans[i-1]=(fft[i-1]*(dum=ans[i-1])-fft[i]*ans[i])/no2;      作FFT相乘以
            ans[i]=(fft[i]*dum+fft[i-1]*ans[i])/no2;                   求卷积
        } else if (isign == -1) {
            if ((mag2=SQR(ans[i-1])+SQR(ans[i])) == 0.0)
                nrerror("Deconvolving at response zero in convlv");
            ans[i-1]=(fft[i-1]*(dum=ans[i-1])+fft[i]*ans[i])/mag2/no2;作FFT相除以
            ans[i]=(fft[i]*dum-fft[i-1]*ans[i])/mag2/no2;              求解卷积
        } else nrerror("No meaning for isign in convlv");
    }
    ans[2]=ans[n+1];                          为realft计算将最后元素压到第一个元素
    realft(ans,n,-1);                         来逆变换返回时域
    free_vector(fft,1,n<<1);
}
```

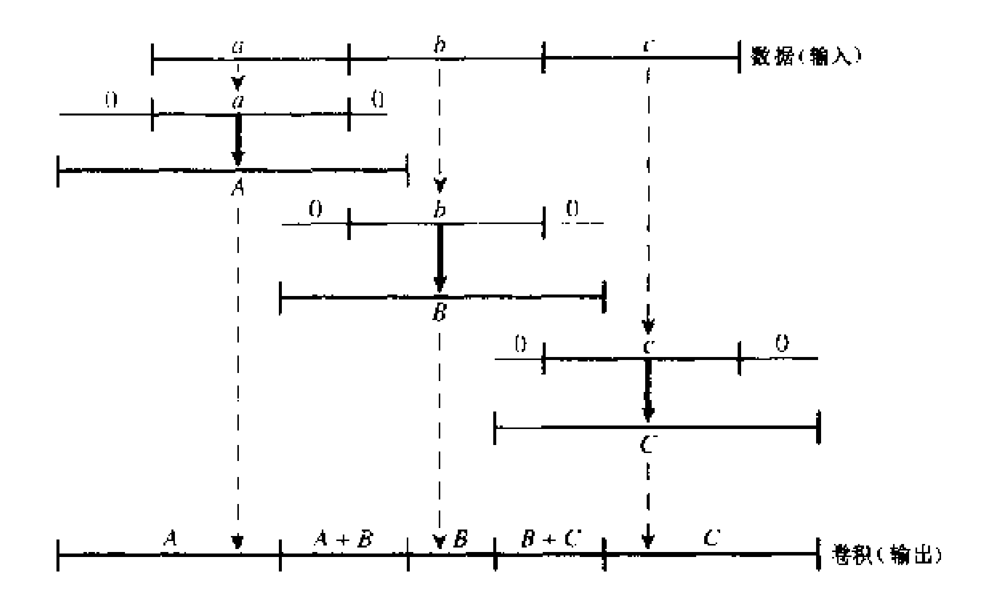
### 13.1.3 大型数据集的卷积和解卷积

如果数据集太长,不能一次全部装入内存,必须将它分成几段并且分别对每段卷积。但是,这时终端效应的处理非常困难。不仅要担心有伪环绕效应,而且还要担心这样的事实,即每段数据的终端肯定受到邻近的前面各段和后面各段之终端数据的影响;但是,因为每次机器内只有一段数据,所以这种所受影响不十分严重。

对于这个问题,有两个相关的、标准的解决方法。这两种方法都是显而易见的,因此这里不多叙述,读者应该能够自己去实现它们。第一种解决方法是**交迭存储法**。在这种技巧中,只需对数据刚开始部分填充足够的零元,以避免环绕污染。初始填充完毕后,应将所有的零元填充都忘掉。取入一段数据,并对它进行卷积或倒卷积,然后丢弃两端上受环绕终端效应污染的点。只输出中部的正常点。接着着取入下一段数据,但不全是新数据。每一新段的前几个点是与前一段数据的后面几点重迭。各段必须充分地交迭,以便重新计算前一段的终端被污染的输出点,它是作为后一段的前几个未被污染的输出点。稍加考虑就可很容易地确定需多少交迭和保存的点。

第二种解决方法称为**交迭相加法**,如图13.1.5所示。此方法不必交迭输入数据。数据的每一段彼此不相交,并且只使用一次。然而,应该仔细地在两端进行零元填充,才能使输出的卷积和倒卷积中没有环绕引起的二重性。然后再将输出端交迭和相加起来。于是,靠近一段

终端附近的输出点，将受到来自于适当地加在下一段起始端上的输入点的影响。同样，对靠近一段起始端附近的输出点也受到如此影响。只是在细节上要作必要的修改。




信号数据被分成若干小段。在每段的两端都填充零元，并作卷积(图中用粗线箭头表示)。最后，各段相加一起，包括由零元填充而形成交迭区域。

图13.1.5　对非常长的信号与响应函数作卷积的交迭相加法

即使在计算机内存够用时，将长数据集分段的方法也能使计算速度略有提高。这是因为 FFT 的 $N\log N$ 次运算比线性 $N$ 稍微慢一些。然而，对数项的变化是如此缓慢，因此人们总是乐于避免交迭相加法和交迭存储法所引起的复杂操作。如果实际工作中需要计算长数据的卷积，还不如直接把全部数据集送入内存，再进行 FFT。这样，便可以有更多的时间去做更有益的事情了。这将在本章随后的各节中叙述。

## 13.2　使用 FFT 作相关和自相关

相关与卷积在数学上有着密切的联系。但是，在某些方面相关更简单一些，因为参与相关的两个函数，并不像参与卷积的数据和响应函数那样在概念上有相异之处。甚至在相关中，函数由不同的但又通常是相似的数据集表示。我们通过直接叠加进行比较，来研究它们的“相关”，并且其中之一作了左移或右移。

我们已经在式(12.0.10)中定义两个连续函数 $g(t)$ 和 $h(t)$ 的相关，它用 $\mathrm{Corr}(g,h)$ 表示，它是一个滞后 $t$ 的函数。我们偶然也用颇不方便的符号 $\mathrm{Corr}(g,h)(t)$ 来明确地表示这种函数与时间依赖性。如果第一个函数($g$)是第二个函数($h$)的严密的复制品，只是前者比后者延迟了某一时间 $t$，即第一个函数位移到第二个函数的右边，则在时间 $t$ 上相关值很大。同样，在某些负值 $t$ 上相关很大，只要第一个函数超前第二个函数 $t$，即位移到第二个函数的左边。这两个函数交换后，得出关系为

$$\mathrm{Corr}(g,h)(t)=\mathrm{Corr}(h,g)(-t) \tag{13.2.1}$$

两个以 $N$ 为周期的样本函数 $g_k$ 和 $h_k$，其离散相关定义为

$$\text{Corr}(g,h)_j \equiv \sum_{k=0}^{N-1} g_{j+k} h_k \tag{13.2.2}$$

离散相关定理表明两个实函数 $g$ 和 $h$ 的离散相关是离散傅里叶变换对中的一个：

$$\text{Corr}(g,h)_j \Leftrightarrow G_k H_k^* \tag{13.2.3}$$

其中 $G_k$ 和 $H_k$ 是 $g_j$ 和 $h_j$ 的离散傅里叶变换，星号表示复共轭。如同离散卷积定理一样，该定理中关于函数要作出一些假设。

我们能够用 FFT 计算相关，其方法如下：对两个数据集进行 FFT，用一个变换结果乘以另一个变换的复共轭，然后再求乘积的逆变换。得出的结果（称为 $r_k$）形式上都是界长为 $N$ 的复向量。然而，因为原两个数据集都是实型的，所以可以证明结果中所有的虚部都为零。$r_k$ 的分量是不同滞后的相关值，正滞后和负滞后都按熟悉的环绕顺序方式存储：零滞后的相关是第一个分量 $r_0$；滞后1的相关是第二个分量 $r_1$；滞后－1的相关是最后一个分量 $r_{N-1}$，如此等等。

如同卷积的情况一样，我们必须考虑终端效应，因为我们的数据通常不象相关定理所期望的那样是周期的。这样我们可以再次使用零元填充。如果对大滞后如$\pm K$感兴趣，则必须在输入数据集的两端加上一个有 K 个零元的缓冲区。如果想要来自 N 个数据点的所有可能的滞后（不是件常见的事），则需要填充与数据相同个数的零元；这是一种极端情况。该程序如下：

```
#include "nrutil.h"

void correl(float data1[], float data2[], unsigned long n, float ans[])
    计算两个实型数据集 data1[1..n]和 data2[1..n]的相关（包括任何为用户提供的零元填充）。n 必须是2的整幂次.
    结果返回到 ans[1..2*n]中前 n 个点，并按环绕次序存储，即逐次递减负滞后，其相关的存储地址从 ans[n]降到
    ans[n/2+1]；而逐次递增正滞后，其相关的存储地址从 ans[1]（零滞后）增加到 ans[n/2]。在程序调用中，数组 ans
    必须具有界长至少为2*n，因为它亦被用作为工作区。本程序的符号约定，如果 data1 滞后 data2，即 data1 位移到 da-
    ta2的右边，则 ans 将在正滞后上呈现峰值。
{
    void realft(float data[], unsigned long n, int isign);
    void twofft(float data1[], float data2[], float fft1[], float fft2[],
        unsigned long n);
    unsigned long no2,i;
    float dum, *fft;

    fft=vector(1,n<<1);
    twofft(data1,data2,fft,ans,n);              立刻变换两个 data 向量
    no2=n>>1;                                   逆 FFT 的归一化
    for (i=2;i<=n+2;i+=2) {
        ans[i-1]=(fft[i-1]*(dum=ans[i-1])+fft[i]*ans[i])/no2;     相乘以求出它们相关
        ans[i]=(fft[i]*dum-fft[i-1]*ans[i])/no2;                  的 FFT
    }
    ans[2]=ans[n+1];                            将第一个和最后一个元素压缩成一个元素
    realft(ans,n,-1);                           求逆变换给出相关
    free_vector(fft,1,n<<1);
}
```

一个样本函数 $g_j$ 的**离散自相关**就是函数与其自身的离散相关。显然，就正、负滞后而言，它总是对称的。人们可以自由地应用以上程序 **correl** 以获得自相关，只要将两个自变量都用同一 data 向量而简单地调用程序。如果因效率低而伤脑筋的话，则当然可以用程序 **realft**

来代替计算 data 向量的 FFT 变换。

## 13.3 具有 FFT 的最佳(维纳)滤波

在数值处理中有许多其它任务，它们在常规情况下都是采用傅里叶技巧处理的。其中之一就是对一个"讹误"信号中的噪声进行滤波。我们研究的情况是这样的：我们要测量一些潜在的、没有被讹误的信号 $u(t)$。但是，由于测量过程不完善，从测量设备得到的信号已是被讹误的信号 $u(t)$，信号 $c(t)$ 可能在两方面或其中的一个方面不够完善。第一，仪器可能不具备完善的"$\delta$ 函数"响应，这样，真实信号被某已知函数 $r(t)$ 所卷积(被 $r(t)$ 涂改)，从而得到一个涂改了的信号 $s(t)$：

$$s(t)=\int_{-\infty}^{\infty} r(t-\tau)u(\tau)d\tau \quad \text{或} \quad S(f)=R(f)U(f) \tag{13.3.1}$$

其中，$S$、$R$、$U$ 分别是 $s$、$r$、$u$ 的傅里叶变换。第二，所测的信号 $c(t)$ 可能包含了噪声 $n(t)$ 的附加分量：

$$c(t)=s(t)+n(t) \tag{13.3.2}$$

我们已经知道，在没有任何噪声的情况下如何对响应函数 $r$ 的效果作解卷积(第13.1节)；我们只需用 $C(f)$ 被 $R(f)$ 除，就得到一个被解卷积的信号。而现在我们要在有噪声的情况下处理类似的问题。我们的任务是寻找**最佳滤波** $\varphi(t)$ 或 $\Phi(f)$，将它们作用于已测信号 $c(t)$ 或 $C(f)$，然后用 $r(t)$ 或 $R(f)$ 与之作解卷积，产生一个信号 $\tilde{u}(t)$ 和 $\tilde{U}(f)$。换言之，我们将用下式来估计真实信号 $U$

$$\tilde{U}(f)=\frac{C(f)\Phi(f)}{R(f)} \tag{13.3.3}$$

$\tilde{U}$ 以什么意义接近 $U$ 呢？我们要求它以**最小二乘方**意义接近 $U$，即

$$\int_{-\infty}^{\infty}|\tilde{u}(t)-u(t)|^2dt=\int_{-\infty}^{\infty}|\tilde{U}(f)-U(f)|^2df \quad \text{为最小} \tag{13.3.4}$$

代入式(13.3.3)和(13.3.2)，式(13.3.4)右边变成

$$\begin{aligned}&\int_{-\infty}^{\infty}\left|\frac{[S(f)+N(f)]\Phi(f)}{R(f)}-\frac{S(f)}{R(f)}\right|^2df\\&=\int_{-\infty}^{\infty}|R(f)|^{-2}\{|S(f)|^2|1-\Phi(f)|^2+|N(f)|^2|\Phi(f)|^2\}df\end{aligned} \tag{13.3.5}$$

信号 $S$ 和噪声 $N$ 是不相关的，所以随频率 $f$ 积分时，它们的叉积得到零。(这就是实际上我们对噪声的定义!)很显然，当且仅当在每个 $f$ 值上被积函数对于 $\Phi(f)$ 为最小，则式(13.3.5)达到极小。让我们来寻找 $\Phi(f)$ 为实函数的这种解。对 $\Phi$ 求偏导，并设其结果为零，即得

$$\Phi(f)=\frac{|S(f)|^2}{|S(f)|^2+|N(f)|^2} \tag{13.3.6}$$

这就是最佳滤波 $\Phi(f)$ 的公式。

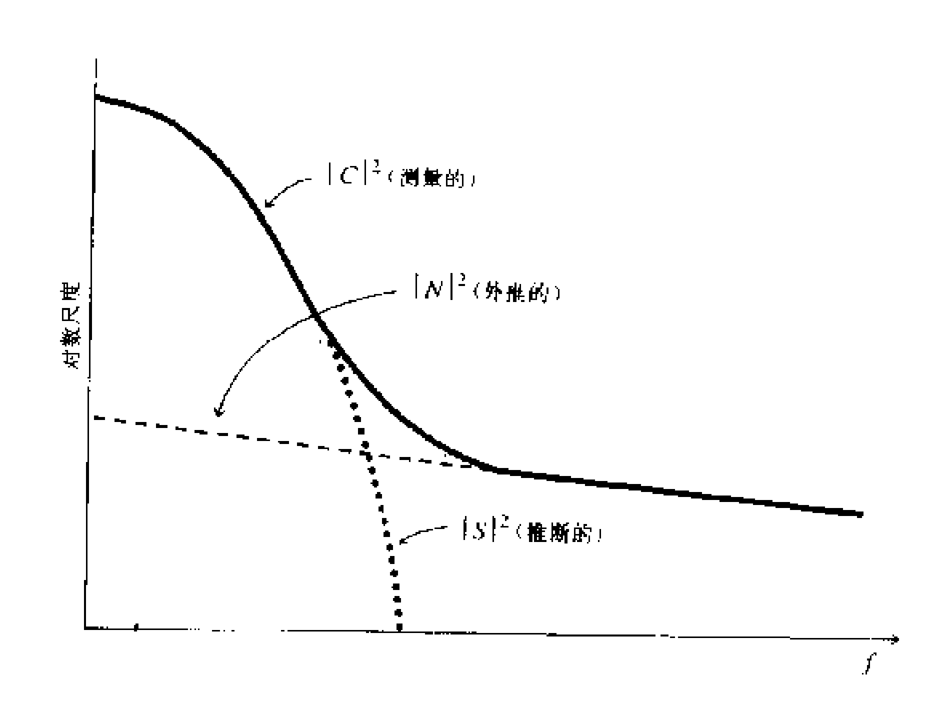
注意，式(13.3.6)中包括已被涂改的信号 $S$ 和噪声 $N$。这两者相加是已测信号 $C$。等式(13.3.6)中不包含"真实"信号 $U$。由此得出一个最重要的简化：最佳滤波的确定，可依赖与 $S$ 和 $U$ 有关的解卷积函数的确定。

为了由式(13.3.6)确定最佳滤波，我们需要一些分别估计 $|S|^2$ 和 $|N|^2$ 的方法。若想只

用已测信号 $C$，而不用一些其它信息，或一些假设、猜想，要做到这点是没有办法的。幸运的是，我们总能容易得到额外的信息。然后用公式(12.0.14)、(12.1.8)和(12.1.5)来绘出来其功率谱密度的图形。这个量是与$|S|^2+|N|^2$成比例，即有

$$|S(f)|^2 + |N(f)|^2 \approx P_c(f) = |C(f)|^2 \quad 0 \le f \le f_c \tag{13.3.7}$$

(在第13.4节和第13.7节中，将讨论更多的估计功率谱密度的复杂方法，但是，上述估计方法总是足以适合解决最佳过滤问题的。)所得结果的图形(如图13.3.1)经常立刻能显示出在连续噪声谱上伸展出的信号之谱特征。噪声谱可能是平坦的，或是倾斜的，或是光滑变化的，这都没有关系，只要我们能够根据它猜测出一个合理的假设。通过噪声谱画一条光滑曲线，将其外推到由信号支配的地方，再通过信号加噪声功率谱处画一条光滑曲线。这两条曲线之差就是信号功率谱的光滑"模型"。信号功率谱模型与信号加功率谱模型之商就是最佳滤波 $\Phi(f)$。[用公式 $\Phi(-f)=\Phi(f)$将其扩展到 $f$ 的负值]。注意，若噪声可忽略之处，则 $\Phi(f)$接近单位1；若噪声占主导地位之处，则 $\Phi(f)$近似零。这就是噪声如何起作用的！由式(13.3.6)给出的中间依赖关系，恰好是得出在这两个极端之间的最佳公式。




信号加噪声的功率谱呈现了一个加上了噪声尾部的信号峰值。这个尾部作为"噪声模型"外推回到信号区域。作减法就得"信号模型"。对有用的方法模型不必是精确的。这些模型的简单代数组合就得到最佳滤波(见正文)。

图13.3.1 最佳(维纳 Wiener)滤波

因为最佳滤波是从求最小值问题中得出的，所以用最佳滤波得出的结果与真实的最佳值相差精度为二阶的数量级，最佳滤波正是按这精度而确定的。换言之，即使是一个相当粗糙的被确定的最佳滤波(例如，在 10% 的水平上)，当应用于数据时，也能给出极好的结果。这就是为什么通常能从一个粗糙的功率谱密度图形中，用"肉眼"将已测得的信号 $C$ 分离成信号分量 $S$ 和噪声分量 $N$ 的原因。这一切都可能会想到对刚才描述的过程进行迭代的问题。例如，用响应 $\Phi(f)$设计一个滤波器，并用它对信号作一合适的猜测 $U(f)=\Phi(f)C(f)/$

$R(f)$以后，再回过来，把$U(f)$看成是一个全新的信号，用同样的过滤技术对它进行估计，作进一步改进。不要将时间浪费在这种思路上。这种结构收敛到$S(f)=0$的信号，的确存在一个收敛的迭代方法，但上述方法不是存在的收敛迭代方法中的一种。

当构造一个最佳过滤器时，可以使用程序 **four1**（第12.2节）或 **realft**（第12.3节）对数据进行 FFT。可以使用第13.1节中描述的方法对数据进行滤波。对于最佳滤波，不需要专用程序 **convlv**，因为需要在频域开始处就构造滤波。但是，如果已经将数据与一已知响应函数作了解卷积，则可在其进行傅里叶逆变换之前，用最佳滤波与它相乘，以此修改程序 **convlv**

## 13.4 使用 FFT 作功率谱估计

在前一节，我们通过取函数$c(t)$的某些有限取样范围的离散傅里叶变换的平方模，来"非正式地"估算了$c(t)$的功率谱密度。在本节中，我们大致要作同样的事情，但是我们更加着重于细节，其结果将会引起一些惊异。

第一个细节是功率谱(亦称作功率密度或 PSD)的归一化。一般说来，函数的平方振幅的测量与 PSD 振幅的测量之间有某种比例关系。不幸的是，在每一种域中，有几种不同描述归一化的约定，并且极易弄错这两种域之间的关系。假设函数$c(t)$在$N$个点取样，产生值为$c_0,\ldots,c_{N-1}$，这些点横跨时间$T$的范围，即$T=(N-1)\Delta$，其中$\Delta$是取样间隔。于是，下面有几种关于总功率的描述：

$$\sum_{j=0}^{N-1}|c_j|^2 \equiv \text{“平方和振幅”} \tag{13.4.1}$$

$$\frac{1}{T}\int_0^T|c(t)|^2dt \approx \frac{1}{N}\sum_{j=0}^{N-1}|c_j|^2 \equiv \text{“均方振幅”} \tag{13.4.2}$$

$$\int_0^T|c(t)|^2dt \approx \Delta\sum_{j=0}^{N-1}|c_j|^2 \equiv \text{“时间积分平方振幅”} \tag{13.4.3}$$

正如我们将要看到的，PSD 估计量甚至有更多的变化。本节中我们考虑其中的一类，它对频率$f_i$的离散值进行估计，其中$i$取于积分值范围内。下一节中，我们将学习不同的另一类估计，即对频率$f$的连续函数进行估计。即便有关 PSD 的归一化，与函数归一化(例如式(13.4.2))的特定描述是一致的话，对 PSD 至少还有下述几种可能性：

- 定义在离散正频率、零频率和负频率上，并且在这些频率上的和是函数的均方振幅
- 仅仅定义在零频率和正频率上，并且在此频率上之和是函数的均方振幅
- 定义在从$-f_c$到$f_c$的奈斯特区间上，并且在此范围内的积分是函数的均方振幅
- 定义在0到$f_c$频率间，在此区域内积分是函数的均方振幅。

在奈奎斯特区间$-f_c$到$f_c$以外对样本函数的 PSD 作积分是毫无意义的。因为根据取样定理，区间以外的功率已被混叠到奈奎斯特区间内。

定义足够多的符号用来区分所有可能的归一化的组合是毫无希望的。因此，以后我们用符号$P(f)$来代表上述 PSD 的任一种定义，并在每一个例子中标出各个$P(f)$是如何被归一化的。要当心文献中那些不相容的符号。

在前一节中使用的功率谱估计方法是一种的简单的估计方法，历史上将这估计量称作为**周期图**。如果我们在相等间隔上取函数$c(t)$的$N$个样本点，并用 FFT 计算它的离散傅里

叶变换

$$C_k = \sum_{j=0}^{N-1} c_j e^{2\pi ijk/N} \quad k = 0,\dots,N-1 \tag{13.4.4}$$

则定义在 $N/2+1$ 个频率上功率谱的周期图为

$$\begin{aligned} P(0) &= P(f_0) = \frac{1}{N^2}|C_0|^2 \\ P(f_k) &= \frac{1}{N^2}\left[|C_k|^2 + |C_{N-k}|^2\right] \quad k = 1,2,\dots,\left(\frac{N}{2}-1\right) \\ P(f_c) &= P(f_{N/2}) = \frac{1}{N^2}|C_{N/2}|^2 \end{aligned} \tag{13.4.5}$$

其中 $f_k$ 只对零频率和正频率定义。

$$f_k \equiv \frac{k}{N\Delta} = 2f_c\frac{k}{N} \quad k = 0,1,\dots,\frac{N}{2} \tag{13.4.6}$$

根据帕斯维尔定理，即方程式(12.1.10)，我们立即看出式(13.4.5)已被归一化，因此，$P$ 的 $N/2+1$ 个值之总和等于函数 $c_j$ 的均方振幅。

现在我们肯定要提这样一个问题，在什么意义下，周期图估计式(13.4.5)是潜在的函数 $c(t)$ 之功率谱的"真实"估计量？这可以在引用的文献中[参见[1]中引言]找到详细的答案，摘要如下。

首先，周期图估计的**期望值**等于功率谱吗？既对又不对。我们确实不会期望 $P(f_k)$ 之一确切地在 $f_k$ 处等于连续的 $P(f)$ 值，因为 $f_k$ 被假设成是整个频率"箱"内的代表，它是从前一个离散频率的半途延伸到下一个离散频率的半途。但是，我们应该期望 $P(f_k)$ 是在以 $f_k$ 为中心的狭窗函数内 $P(f)$ 的某种类型的平均值。对于周期图估计式(13.4.6)，其窗函数，作为箱中频率偏移 $s$ 的函数，是

$$W(s) = \frac{1}{N^2}\left[\frac{\sin(\pi s)}{\sin(\pi s/N)}\right]^2 \tag{13.4.7}$$

注意 $W(s)$ 是振荡波瓣，除此以外，它大约以 $W(s)\approx(\pi s)^{-2}$ 衰减。这不是一个快速衰减，其导致周期图的估计中从一个频率到另一频率的重要渗漏(这是一个技术术语)。还要注意，当 $s$ 为非零整数时，$w(s)$ 恰好为零。这意味着如果函数 $c(t)$ 正好等于 $f_k$ 中某一个频率的纯正弦波，则相邻 $f_k$ 之间就不会有渗漏。但这不是典型的情况！如果频率在两个相邻 $f_k$ 之间的1/3处，则渗漏将大大超过这两个相邻的箱。解决渗漏问题的方法称为**数据窗**，下面将对其进行讨论。

现在我们转向关于周期图估计的另一个问题。随 $N$ 趋于无穷时估计的方差有多大？换言之，当随着对原始函数取样点的增点(或者以相同的取样率取样一个更长的数据，或者以更快的取样率对相同的数据重新取样)，估计 $P_k$ 的正确性将增加多少？答案是令人失望的，周期图估计根本不会变得更精确！事实上，频率 $f_k$ 处的周期图估计的方差始终等于在该频率处期望值的平方。换言之，标准差永远是该值的100%，与 $N$ 无关！这可能是怎么回事？我们增加点后的所有信息到哪儿去了？它们都参与了更大量的离散频率 $f_k$ 的估计过程中去了。如果我们用同样的取样率对一个更长的数据取样，则奈奎斯特临界频率 $f_c$ 是不变的，不过现在在奈奎斯特频率区内有更细的频率分解(更多的 $f_k$)；另外，若用更细的取样间隔对同样长度的数据取样，则频率分解不变，但现在奈奎斯特区域扩展至更高的频率。在这两种

情况下，增加样本都不会减少任一特定频率估计的 PSD 的方差。

不管怎样，是不必承受百分之百的标准差的 PSD 估计。只要知道几种减少估计方差的技巧就可以了。这里有两种技巧，它们在数学上几乎等同，但是执行过程不同。第一种是，采用比实际所需还要更细一些的离散频率间隔进行取样并计算周期图估计，然后对 $k$ 个相邻的离散频率上求周期图估计的和，以得到在这样 $K$ 个频率的中间频率上的一个"更光滑"的估计。这个和估计的方差比其估计本身小一个$1/K$ 因子，即标准差比 100% 小一个 $1/\sqrt{K}$ 因子。因此，为了对 0 和 $f_c$ 之间 $M-1$个离散频率上估计功率谱，就开始需取$2MK$ 点的 FFT（这个数 $2MK$ 最好是 2 的整数次幂）。然后取结果系数的模平方，将正的和负的频率对加起来，再被$(2MK)^2$除，所有这些都是按照式(13.4.5)并取 $N=2MK$ 进行的。最后，将结果"装箱"，使之成为 $k$ 的求和组（不是平均组）。这过程很容易编程，因此不必为没给出程序而烦恼。对 $K$ 个相邻点求和而不是求平均，其原因是为了使最终 PSD 估算将保持其 $M-1$ 个值的和等于函数均方振幅的归一化特性。

在区间0到 $f_c$ 之间，估计 $M+1$个频率上的 PSD 的第二种技巧是，将原始样本数据划分成 $K$ 个小段，每段有 $2M$ 个连续的样本点。每段分别作 FFT，以得到一个周期图估计（式(13.4.5)且取 $N=2M$）。最后，在每个频率上对 $K$ 个周期图估计求平均。正是最后一步平均，使估计的方差降低了一个 $K$ 因子（标准差降低了 $\sqrt{K}$因子）。从计算上看，第二种技巧比前面第一种技巧更为有效，因为对较短样本点作 FFT 比对较长样本点作 FFT，其效率按对数量级提高。不过，第二种技巧的主要优点还在于，在一段时间内只处理 $2M$ 个数据点，而不象第一种技巧中是处理 $2MK$ 个数据点。这意味着，对来自磁带或其它数据记录中的长数据的处理，第二种技巧是一种自然的选择。后面我们将给出实施第二种技巧的程序。但是，我们还需首先返回到渗漏和由式(13.4.7)引出的数据开窗问题中来。

### 13.4.1 数据开窗

数据开窗之目的在于修正公式(13.4.7)式，它表示一个离散频率的谱估计 $P_k$ 与相近频率上实际的、潜在的连续谱 $P(f)$之间的关系。一般说来，一个"量化层" $k$ 内的功率谱实际上包括着其它 $s$ 量化层的频率分量所带来的渗漏，其中 $s$ 是式(13.4.7)中的自变量。正如我们所指出的，甚至从适当大的 $s$ 值，也有大量的渗漏。

当我们为周期谱估计选择一组 $N$ 个样本点时，实际上已用了一个时间的窗函数乘上一个无穷长的样本数据据组 $c_j$，这窗函数在整个取样时间 $N\Delta$ 内等于单位值，而其它时间取值为零。换言之，数据被一个矩形窗函数开了窗。根据卷积定理（式(12.0.9)；但是互换 $f$ 和 $t$ 的位置），数据与这个矩形窗函数乘积的傅里叶变换，应该等于数据的傅里叶变换与窗函数的傅里叶变换的卷积。事实上，我们确定的方程式(13.4.7)，并不比单位窗函数的离散傅里叶变换的平方多点什么。

$$W(s)=\frac{1}{N^2}\left[\frac{\sin(\pi s)}{\sin(\pi s/N)}\right]^2=\frac{1}{N^2}\left|\sum_{k=0}^{N-1}e^{2\pi isk/N}\right|^2 \qquad (13.4.8)$$

在 $s$ 大值处，发生渗漏的原因在于，矩形窗函数接通和断开太迅速，它的傅里叶变换在高频处有大量的分量。为了改善这种情况，用窗函数乘输入数据 $c_j(j=0,\ldots,N-1)$，使窗函数 $w_j$ 随 $j$ 从 0 至 $N$ 变化而从零逐渐升至极大值，然后再降回到零。在这种情况下，周期图估计量（式(13.4.4~13.4.5)）成为：

$$D_k \equiv \sum_{j=0}^{N-1} c_j w_j e^{2\pi i jk/N} \quad k=0,\ldots,N-1 \tag{13.4.9}$$

$$P(0)=P(f_0)=\frac{1}{W_{ss}}|D_0|^2$$

$$P(f_k)=\frac{1}{W_{ss}}\left[|D_k|^2+|D_{N-k}|^2\right] \quad k=1,2,\ldots,\left(\frac{N}{2}-1\right)$$

$$P(f_c)=P(f_{N/2})=\frac{1}{W_{ss}}|D_{N/2}|^2 \tag{13.4.10}$$

其中$W_{ss}$代表“平方求和之窗”

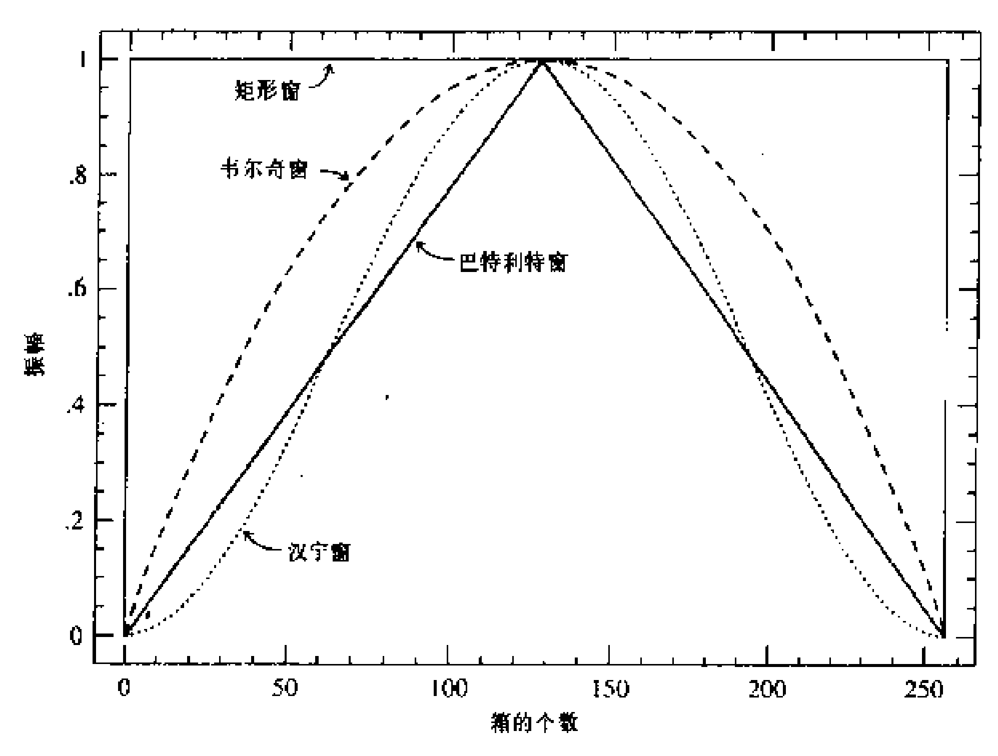
$$W_{ss}\equiv N\sum_{j=0}^{N-1} w_j^2 \tag{13.4.11}$$

其中$f_k$由式(13.4.6)给出。式(13.4.7)更一般的形式现在可以根据窗函数$w_j$写成：

$$\begin{aligned} W(s) &= \frac{1}{W_{ss}}\left|\sum_{k=0}^{N-1} e^{2\pi i sk/N} w_k\right|^2 \\ &\approx \frac{1}{W_{ss}}\left|\int_{-N/2}^{N/2} \cos(2\pi s\, k/N)\, w(k-N/2)dk\right|^2 \end{aligned} \tag{13.4.12}$$

此处的近似式对实际估计是很有用的，而且适用于任何左右对称的窗（通常的情况）和对$s<<N$的情况（这是估计渗漏进入邻近量化层的情况）。积分中的连续函数$w(k-N/2)$是通过点$w_k$的某光滑函数。

关于选择窗函数，也许有许多不必要的经验，实践中，每个从零开始上升到顶峰然后又下降的函数都是以某个人的名字命名。常用的几个是如下（见图13.4.1所示）：




在FFT计算之前，将数据段乘窗函数（依此逐个量化层），数据段的界长为256。矩形窗，它等价于没有开窗，是不值得推荐的。韦尔奇窗和巴特利特窗是很好的选择。

图13.4.1　通常用于FFT功率谱的窗函数

$$w_j = 1 - \left|\frac{j-\frac{1}{2}N}{\frac{1}{2}N}\right| \equiv \text{“巴特利特窗”} \tag{13.4.13}$$

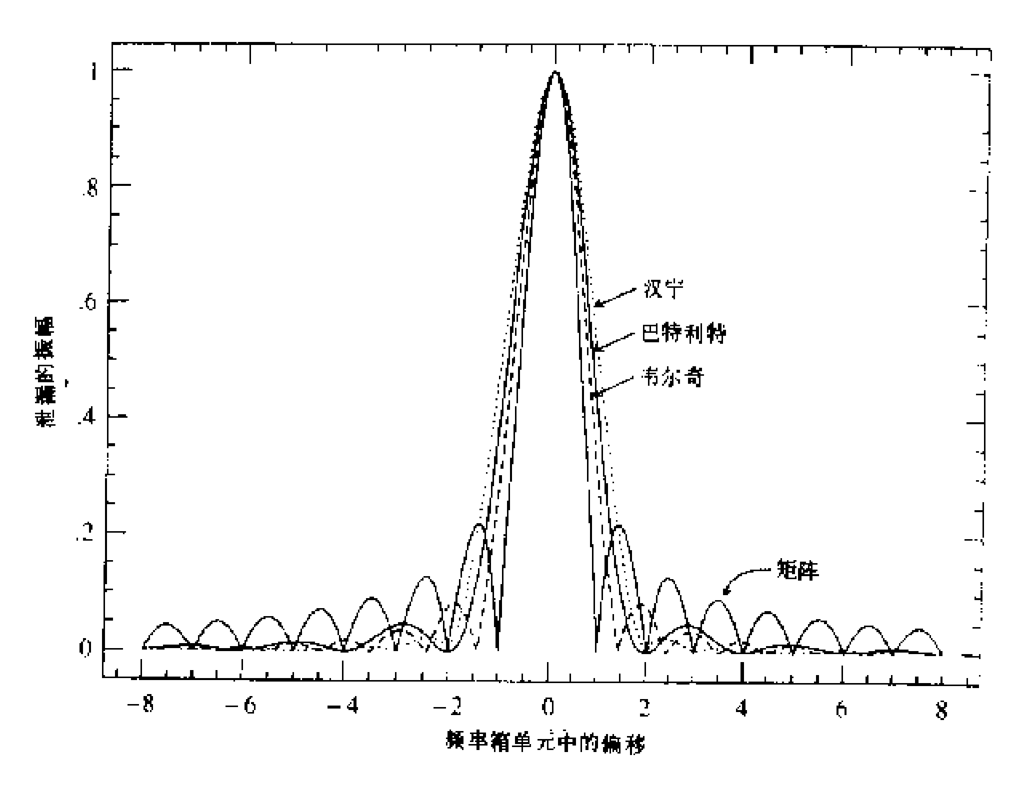
(“帕森(Parzen)窗”与此非常类似)。

$$w_j = \frac{1}{2}\left[1-\cos\left(\frac{2\pi j}{N}\right)\right] \equiv \text{“汉宁(Hann)窗”} \tag{13.4.14}$$

(“汉明(Hamming)窗”是类似的,但其在端部不会准确地到达零)。

$$w_j = 1 - \left(\frac{j-\frac{1}{2}N}{\frac{1}{2}N}\right)^2 \equiv \text{“韦尔奇(Welch)窗”} \tag{13.4.15}$$

我们倾向于遵循韦尔奇,推荐读者在实际工作中使用式(13.4.13)或者式(13.4.15)。然而,以本书的水平,任何这些(或类似的)窗函数没有实在的区别。它们之间的差别在于各种有价值的标准图之间的微妙的权衡,这些标准图形可以用来描述由式(13.4.12)计算出的谱渗漏函数的狭窄性和有峰性。这些标准图形还有一些名称:**最高旁瓣**(db)、**旁瓣衰减**(**每个倍频程的** db)、**等价噪声带宽**(**量化层**)、**3-db 带宽**(**量化层**)、**扇状损失**(db)、**最坏情况过程损失**(db)。粗略地说,是在形成尽可能狭窄的中间峰值区和尽可能加快尾部的衰减两者之间进行主要的权衡。详情参阅文献[2]。图13.4.2中绘出了已讨论过的几种窗的渗漏振幅图。




实际上频率位于零偏移处的一个信号“渗漏”到具有图示振幅的邻近量化层中。开窗的目的是为了减少在大频率偏移处的渗漏,而矩形窗(不开窗)具有大的旁瓣。偏移可以是分数值,因为实际信号频率可能位于 FFT 的两个频率量化层之间。

图13.4.2　图13.4.1中窗函数的渗漏函数

关于窗函数，还有一些经验知识.在数据的前一小部分处(设 10% )窗函数从零平滑地上升到单位值，然后窗函数停留在单位值，直到数据最后的一小部分(又假设为 10% )，在这期间函数又从单位值平滑地下降到零。这个窗函数虽然使渗漏函数的主瓣变窄一点(永不会如 2 的倍数那么多)，但将其渗漏尾部拓宽一个明显的因子(例如，10% 的倒数，一个 10 的因子)。假如我们能将窗的宽度(在窗的极大值处的样本个数)和窗的上升(下降)时间(在其上升与下降过程中的样本数)区分开来;如果我们又能将渗漏函数主瓣的 FWHM(半最大值的全宽度)和**渗透宽度**(包含谱功率一半的全宽度，这些功率谱不包含在主瓣内)区分开来;则这些量粗略地联系起来为

$$(\text{量化层中 } FWHM) \approx \frac{N}{(\text{窗宽})} \tag{13.4.16}$$

$$(\text{量化层中渗漏宽度}) \approx \frac{N}{(\text{窗上升(下降)时间})} \tag{13.4.17}$$

对于前面式(13.4.13)～(13.4.15)给出的窗，有效窗宽和有效上升(下降)时间都是 $\frac{1}{2}N$ 阶的。通常而言，我们觉得，当窗的上升和下降时间只是数据界长的很小部分，这种窗优点很小或没有时，我们避免使用它们。有时可能会听说，平顶的窗"损失较少数据"，但是我们现在告诉读者一个更好的办法来解决这个问题，即用交迭数据段的方法。

假设选择了一个窗口函数，并准备把数据划分为有 $N=2M$ 个点的 $K$ 段。对每段作 FFT，并将得到的 $K$ 个周期图一起求平均，以获取 0 到 $f_c$ 间的 $M$ 个频率值上的 PSD.现在我们必须区分两种可能的情况。我们可能想从固定的计算总数中得到最小方差，而不必须考虑所用数据点的个数。例如，通常当数据正在实时地聚集，并且数据的简化受计算机控制时，这种情况便成为目标。另一种可能情况是，我们可能想从有效的样本数据点中得到最小方差。通常，如果数据已经记录下来，并记录之后正在分析，则这便成为目标。

在第一种情况下(每次计算操作的最小谱方差)，最好不交迭而将数据分段。前 $2M$ 数据点构成1号段;其次 $2M$ 个数据点构成 2 号段;以此类推，直至 $K$ 号段，共有 $2KM$ 个样本点。这种情况下的方差和单个段有关，方差减少到 $1/K$ 。

第二种情况下(每个数据点的最小谱方差)，若将各段交迭长度的一半，则所得的是最佳的或非常接近最佳的。第一值 $M$ 个点和第二组 $M$ 个点为 1 号段;第二组 $M$ 个点又和第三组 $M$ 个点为 2 号段;以此类推，直至 $K$ 号段，它由第 $K$ 组 $M$ 个点和 $K+1$组 $M$ 个点组成。从而样本点总数是$(K+1)M$，正好是没有用交迭段的总数之半。方差的减少不是 $1/K$ 的整数倍，因为各段不是统计独立的。可以证明，方差大约减少 $11/9K$(见韦尔奇的文章[3])。无论怎样，这比减少 $2/K$ 要好;如果没有交迭，用同样数量进行分段，这就降低 $2/K$。

现在将这些想法编制成一个谱估计程序。通常，我们避免输入(输出)编码，但这里例外，将呈现如何通过一个数据文件顺序读出数据(参见参数 FILE * fp)。在任一时刻，只有一小部分数据在内存中。

```
#include <math.h>
#include <stdio.h>
#include "nrutil.h"
#define WINDOW(j,a,b) (1.0-fabs((((j)-1)-(a))*(b))) /* Bartlett */
/* #define WINDOW(j,a,b) 1.0 */ /* Square */
```

```
/* #define WINDOW(j,a,b) (1.0-SQR((((j)-1)-(a))*(b))) */ /* Welch */

void spctrm(FILE *fp, float p[], int m, int k, int ovrlap)
    由文件指针fp指定的输入流中读出数据，并且将每个网格点频率(j-1)/(2*m)周期处的数据之功率谱(均方根振
    幅)返回到p[j](j=1,2,...,m)。这些网格点是基于(2*k+1)*m个数据点(如果设ovelap是真(1));或者基于4*
    k*m个数据点(如果设ovrlap是假(0))。这两种情况数据段的个数都是2*k;程序中调用four1 k次，每次调用有
    两个部分，每部分有2*m个实数据点
{
    void four1(float data[], unsigned long nn, int isign);
    int mm,m44,m43,m4,kk,joffn,joff,j2,j;
    float w,facp,facm,*w1,*w2,sumw=0.0,den=0.0;

    mm=m+m;                                       有用的式
    m43=(m4=mm+mm)+3;
    m44=m43+1;
    w1=vector(1,m4);
    w2=vector(1,m);
    facm=m;
    facp=1.0/m;
    for (j=1;j<=mm;j++) sumw += SQR(WINDOW(j,facm,facp));
    累加权重的平方和
    for (j=1;j<=m;j++) p[j]=0.0;                将谱初始赋值为零
    if (ovrlap)                                 对"保存"的半缓冲区初始赋值
        for (j=1;j<=m;j++) fscanf(fp,"%f",&w2[j]);
    for (kk=1;kk<=k;kk++) {
    以两组数据集的段做循环
        for (joff = -1;joff<=0;joff++) {        两个完备段进入工作区
            if (ovrlap) {
                for (j=1;j<=m;j++) w1[joff+j+j]=w2[j];
                for (j=1;j<=m;j++) fscanf(fp,"%f",&w2[j]);
                joffn=joff+mm;
                for (j=1;j<=m;j++) w1[joffn+j+j]=w2[j];
            } else {
                for (j=joff+2;j<=m4;j+=2)
                    fscanf(fp,"%f",&w1[j]);
            }
        }
        for (j=1;j<=mm;j++) {                   将窗应用于数据
            j2=j+j;
            w=WINDOW(j,facm,facp);
            w1[j2] *= w;
            w1[j2-1] *= w;
        }
        four1(w1,mm,1);                         对开窗的数据做FFT变换
        p[1] += (SQR(w1[1])+SQR(w1[2]));        结果之和进入原先的段
        for (j=2;j<=m;j++) {
            j2=j+j;
            p[j] += (SQR(w1[j2])+SQR(w1[j2-1])
                +SQR(w1[m44-j2])+SQR(w1[m43-j2]));
        }
        den += sumw;
    }
    den *= m4;                                   正确的规一化
    for (j=1;j<=m;j++) p[j] /= den;             将输出规一化
    free_vector(w2,1,m);
    free_vector(w1,1,m4);
}
```

## 参考文献和进一步读物：


Oppenheim, A. V., and Schafer, R. W. 1989, *Discrete-Time Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Harris, F.J. 1978, *Proceeding of the IEEE*, vol. 66, pp. 51~83. [2]

Childer, D.G. (ed.) 1978, *Modern Spectrum Analysis* (New York:IEEE Press), paper by P.D.Welch. [3]

Rabiner, L.R., and Gold, B. 1975. *Theory and Application of Digital Signal Processing* (Englewood Cliffs, NJ:Prentice-Hall).

## 13.5 时域中的数字滤波

假设有一个信号要进行数字滤波。例如，也许想运用**高通**或**低通滤波**，来分别滤去低频或高频噪声；也许只对处于某个频带内的信号感兴趣，则需要**带通**滤波器。或者，如果测量中混杂有 60Hz 电源线干扰，就可能需要一个**槽型滤波器**，恰好除去围绕该频率的狭窄带状。这一节特别阐述在时域中，不得不选择这种滤波的情况。

继续讨论前，我们希望读者再考虑一下这种选择。回忆一下，在傅里叶域中滤波是非常方便的。只需记录全部数据，对其进行 FFT，将 FFT 的输出乘以一个滤波函数 $\mathscr{H}(f)$，然后再作逆 FFT，得到一个返回时域中已滤波的数据集。下面是一些考虑的傅里叶技巧的附加背景。

- 请记住，必须定义滤波函数在正、负频率区上，而且频率端部的量值始终是奈奎斯特频率 $1/2\Delta$，其中 $\Delta$ 是取样区间。FFT 中最小非零频率的数量值应为 $\pm 1/(N\Delta)$，其中 $N$ 是 FFT 中（复型）点的个数。这个滤波的正、负频率按环绕顺序排列。
- 如果已测数据是实数，而且希望滤波输出也是实数，则任意滤波函数都应遵循 $\mathscr{H}(-f)=\mathscr{H}(f)^*$。可以极容易地实现这点，只需取一个关于 $f$ 是实并为偶的 $\mathscr{H}$。
- 如果选择的 $\mathscr{H}(f)$ 有锐利垂直的边界，则滤波器的脉冲响应（输入一个短脉冲引起的输出）在边界对应的频率处成“环形”衰减。这并没有出错，但如果不喜欢，那么就选择一个更平滑的 $\mathscr{H}(f)$。为了直接得到关于滤波器脉冲响应的第一手材料，可以取 $\mathscr{H}(f)$ 的逆 FFT。如果在某些 $k$ 个点上对滤波函数所有的边界作了光滑处理，则滤波器的脉冲响应函数将有一个跨度整个数据记录的 $1/k$ 的间距。
- 如果数据太长，不能一次进行 FFT，则需将它们分成适当长度的若干段，只要每度长度比滤波器的脉冲响应函数长得多就行。必要时，使用零元填充。
- 可能需要从数据中减去一条通过第一点和最后一点的一条直线（即，使第一点和最后一点为零），从而除掉来自数据任何趋势。如果正在进行数据划分成段，则可以选取交迭的段，并只使用每段的中间部分，就能相当满意地远离边缘效应。
- 如果一个数字滤波器的某特定时间步的输出仅仅依赖于这个时间步的输入或者更早时间步的输入，则称这个滤波器为**因果性的**或**物理可实现的**。如果它的输出既依赖于早先的输入、也依赖于以后的输入，则称它为**无因果性的**。一般情况，傅里叶域中的滤波器是无因果性的，因为数据是“成批”进行处理的，并不考虑时间顺序。别让这点给难住！通常无因果性的滤波器能带来高级的性能（例如，较少的相位色散，更尖锐的边缘，脉冲响应函数的不对称性减少）。人们使用因果性的滤波器不是因为它们更好些，而是因为在一些场合下不允许存取非时序的数据。原则上讲，时间滤波器既可为因果性的，也可为无因果性的，但是它们通常被用于物理可实现性受限的应用中。按此道理，在以下内容中我们限定于有因果性的情况。

如果读者仍然偏爱时域滤波，这大概因为你有一个实时的应用问题，因此需要处理一串连续的数据流，而且希望按照接收原始数据的相同的速率输出滤波值。否则，可能是待处理的数据量太大，以致于只能在每个数据点上承受非常小量的浮点运算，而且甚至不能进行中规范大小的 FFT（每个数据点的浮点运算次数几倍于数据集或段中点的个数的对数值）。

## 13.5.1 线性滤波器

最常见的线性滤波器其输入点序列为 $x_k$，并由下述公式产生输出点序列 $y_n$

$$y_n = \sum_{k=0}^{M} c_k x_{n-k} + \sum_{j=1}^{N} d_j y_{n-j} \tag{13.5.1}$$

其中 $M+1$ 个系数 $c_k$ 和 $N$ 个系数 $d_j$ 都是固定的，并定义了滤波器的响应。式(13.5.1)滤波器是由当前输入值和先前 $M$ 个输入值，以及它自己先前 $N$ 个输出值产生每个新的输出值。如果 $N=0$，式(13.5.1)中第二项不存在，则滤波称为**非递推滤波或有限脉冲响应滤波**($FIR$)。如果 $N\neq0$，则滤波称为**递推的或无限脉冲响应的**($IIR$)。(术语"$IIR$"仅仅意味着这种滤波器能够具有无限长的脉冲响应，而不是说在特殊的应用中脉冲响应必定很长。典型的情况是，一个 $IIR$ 滤波器的响应在以后时间里将成指数衰减，迅速衰减而变成可以忽略不计。)

$c_k$ 和 $d_j$ 与滤波的响应函数 $\mathscr{H}(f)$ 的关系为

$$\mathscr{H}(f) = \frac{\sum_{k=0}^{M} c_k e^{-2\pi i k(f\Delta)}}{1 - \sum_{j=1}^{N} d_j e^{-2\pi i j(f\Delta)}} \tag{13.5.2}$$

其中 $\Delta$ 仍为取样频率。奈奎斯特区间对应于 $-1/2$ 至 $1/2$ 之间的 $f\Delta$。对 FIR 滤波来说，式(13.5.2)的分母恰好是单位 1。

式(13.5.2)告诉了我们如何由一系列 $c$ 和 $d$ 值确定 $\mathscr{H}(f)$。然而，为了设计滤波，我们需要一个进行逆变换的方法，即从期望的 $\mathscr{H}(f)$ 中获得一个适当的 $c$ 和 $d$ 的集合，这个集要尽可能地小，以使计算负担最小。有许多书整本地致力于这个问题。象其它许多"逆问题"一样，它没有完美的解决方法。因为 $\mathscr{H}(f)$ 是一个完全连续的函数，而小段的 $c$ 和 $d$ 只表示一些可调整的参数，因此人们必须折衷一下。滤波器设计的问题关系到作这种折衷的各类方法。我们不希望能给出一种彻底的处理方法，但是我们可以概述一些基本技巧促使这一工作可以开始，有关进一步详细讨论，必须查阅某些专门化书籍(参见文献)。

## 13.5.2 $FIR$(非递推)滤波

当式(13.5.2)中分母为单位1时，右端恰好是离散傅里叶变换。变换很容易求逆，根据在某些离散频率 $f_i$ 上 $\mathscr{H}(f)$ 的少量几个值，能求得同样少量几个欲求的 $c_k$ 系数。然而，这一事实并非十分有用，因为，按这种方法计算出的 $c_k$ 值，$\mathscr{H}(f)$ 将在离散频率间趋于剧烈地振荡，而在这些离散频率上取特定值。

一个更好的策略，文献中几种正规方法的基础，是这样的：开始时，极力找到相对大量滤波系数，即，相当大的 $M$ 值。然后，$\mathscr{H}(f)$ 可以在相当细的网格上固定为所期望的值，并且 $M$ 个系数 $c_k(k=0,\ldots,M-1)$ 可由 FFT 找到。下一步，将大部分 $c_k$ 截断(置为零)，留下非零 $c_k$，例如前 $K$ 个系数($c_0,c_1,\ldots,c_{K-1}$)和最后 $K-1$ 个系数($c_{M-K+1},\ldots,c_{M-1}$)。因为 FFT 的环绕性质，最后少数几个系数 $c_k$ 是负滞后的滤波系数，但是我们并不要负滞后的系数，所以我们循环地位移 $c_k$ 数组，以使得每个系数都成为正滞后(这相当于将时间延迟引入滤波器)。要做到这点，可通过按以下顺序将 $c_k$ 系数复制成一组 $M$ 长的新数组：

$$(c_{M-K+1},\ldots,c_{M-1},c_0,c_1,\ldots,c_{K-1},0,0,\ldots,0) \tag{13.5.3}$$

为了观察所做的截断是否被接受，可以对式(13.5.3)的数组作 FFT，给出原来 $\mathscr{H}(f)$ 的近似值。一般情况、将模 $|\mathscr{H}(f)|$ 与原来的函数作比较，因为时间延迟已将复相位引入滤波响应中了。

如果新的滤波函数是可被接受的，则已进行完毕，并且有了一组 $2K-1$ 个滤波系数。如果它不可被接受，则既可以 (i)增大 $K$ 再试一次，又可以 (ii)对同样的 $K$，进行一些更仔细的工作以提高其可接受性。一个有关更仔细工作的例子就是修改不可接受的 $\mathscr{H}(f)$ 的振幅(而不是相位)，以使它与理想的更一致，然后进行 FFT 得到新的 $c_k$。再次全置零除了前 $2K-1$ 个值(不需要循环位移，因为已保护了时间延迟的相位)，然后求逆变换以得到新的 $\mathscr{H}(f)$，它往往是可被接受的。可以对这过程进行迭代。请注意，如果对接受性的

要求过于苛刻，超过 $2K-1$ 个系数所能操纵的范围，则这个迭代过程不收敛。

换言之，关键思想是在于，在系数空间与函数 $\mathscr{H}(f)$ 空间两者之间进行迭代，直到找到一对傅立叶共轭对，它们在两空间中都满足所强加的限制为止。这种迭代的一种更正规的技术是 **Remes 交换算法**。该算法对具有固定个数滤波系数的所期望的频率响应将产生最佳的切比雪夫逼近（参阅第5.13节）。

### 13.5.3 $IIR$（递推）滤波器

递推滤波在某一给定的时间上，其输出既取决于当前输入和以前的输入，也取决于以前的输出。通常它比具有相同数量系数的非递推滤波更有优势。通过式(13.5.2)检验，理由是非常清楚的：非递推滤波具有一个与变量$1/z$ 成多项式的频率响应，其中

$$z \equiv e^{2\pi i(f\Delta)} \tag{13.5.4}$$

而相比较，递推滤波的频率响应是变量$1/z$ 的**有理函数**，而有理函数对于拟合具有锐利边缘或狭窄特征的函数特别良好，并且大多数期望的滤波函数都是这种类型的。

非递推滤波永远是稳定的。如果将输入的新 $x_i$ 序列断开，则不超过 $M$ 步以后，由式(13.5.1)产生的 $y_i$ 序列也会断开。而递推滤波，如象他们在进行输出时又反馈给自己那样，不必须是稳定的。如果系数 $d_j$ 选得很糟，则递推滤波就可能成指数增长的方式——所谓**齐次的**，即使输入序列已经断开，输出序列也变得很大。这是很不好的。所以，设计递推滤波的问题并不仅仅是一个逆问题，这还是一个具有附加稳定性约束条件的逆问题。

对一组给定的 $c_k$ 和 $d_j$ 系数，如何得知式(13.5.1)的滤波器是否是稳定的呢？稳定性仅仅取决于 $d_j$。当且仅当**特征多项式**

$$z^N - \sum_{j=1}^{N} d_j z^{N-j} = 0 \tag{13.5.5}$$

的所有 $N$ 个复根都在单位圆内时，滤波器是稳定的，即满足

$$|z| \le 1 \tag{13.5.6}$$

构造稳定递推滤波器的各种方法形成了一个学科领域，这需要更多的专门书籍。然而，有一个非常有用的技术是**双线性变换方法**。关于这一专题，我们定义一个新变量 $w$，它将频率 $f$ 重新参数化

$$w \equiv \tan[\pi(f\Delta)] = i\left(\frac{1 - e^{2\pi i(f\Delta)}}{1 + e^{2\pi i(f\Delta)}}\right) = i\left(\frac{1-z}{1-z}\right) \tag{13.5.7}$$

不要被式(13.5.7)中的 $i$ 所欺骗。这一等式把实频率 $f$ 映射成实值 $w$。事实上，它把奈奎斯特区间 $-\frac{1}{2}<f\Delta<\frac{1}{2}$ 映射成实 $w$ 轴 $-\infty<w<+\infty$。式(13.5.7)的逆等式为

$$z = e^{2\pi i(f\Delta)} = \frac{1+iw}{1-iw} \tag{13.5.8}$$

在重参数化 $f$ 时，当然 $w$ 也再次参数化 $z$，所以，稳定的条件(13.5.5)、(13.5.6)可按 $w$ 重新描述：如果滤波响应 $\mathscr{H}(f)$ 写成 $w$ 的函数，则当且仅当滤波函数的极点（其分母的零点）都在上半复平面

$$\text{Im}(w) \ge 0 \tag{13.5.9}$$

时，滤波是稳定的。

双线性变换方法的思路是，只需指定所期望的滤波函数的模平方 $|\mathscr{H}(f)|^2 = \mathscr{H}(f)\mathscr{H}(f)^* = \mathscr{H}(f)\mathscr{H}(-f)$，来代替指定所期望的 $\mathscr{H}(f)$。选取由一些 $w^2$ 的有理函数逼近的函数，然后在 $w$ 复平面上找出这个函数的所有极点。由于对称性，下半平面的每个极点都在上半平面上有一个对应的极点。其思路是形成只以好的极点——在上半平面的极点——为因子的乘积。这个乘积就是**稳定可实现的** $\mathscr{H}(f)$。现在，代入式(13.5.7)将函数写成为 $z$ 的有理函数，并与式(13.5.2)比较，读出系数 $c$ 和 $d$。

通过下面例题的讨论，这一过程会更清楚。假设我们要设计一个简单的带通滤波器，其低截止频率对应于值 $w=a$，而高截止频率对应于 $w=b$ 值，$a$、$b$ 均为正数。能够完成这要求的一个简单有理函数是

$$|\mathscr{H}(f)|^2 = \left(\frac{w^2}{w^2+a^2}\right)\left(\frac{b^2}{w^2+b^2}\right) \tag{13.5.10}$$

这个函数并没有非常突变的截止，但它阐明了更普通的情况。为得到更陡峭的边缘，人们可以取函数(13.5.10)为某个正整数幂次，或等价地，通过从式(13.5.13)中得到的滤波器的某些复制，来连续地运行数据。

式(13.5.10)的极点明显地都在 $w=\pm ia$ 和 $w=\pm ib$ 上。所以稳定可实现的 $\mathscr{H}(f)$ 是

$$\mathscr{H}(f) = \left(\frac{w}{w-ia}\right)\left(\frac{ib}{w-ib}\right) = \frac{\left(\frac{1-z}{1+z}\right)b}{\left[\left(\frac{1-z}{1+z}\right)-a\right]\left[\left(\frac{1-z}{1+z}\right)-b\right]} \tag{13.5.11}$$

我们把 $i$ 放入第二个因式的分子中，为的是能得到实值的系数。如果我们将分母乘出来，式(13.5.11)可写成形式为

$$\mathscr{H}(f) = \frac{-\frac{b}{(1+a)(1+b)} + \frac{b}{(1+a)(1-b)}z^{-2}}{1-\frac{(1+a)(1-b)+(1-a)(1+b)}{(1-a)(1-b)}z^{-1}-\frac{(1-a)(1-b)}{(1+a)(1+b)}z^{-2}} \tag{13.5.12}$$

由此，人们可以读出式(13.5.1)的滤波系数

$$\begin{aligned}
c_0 &= -\frac{b}{(1+a)(1+b)} \\
c_1 &= 0 \\
c_2 &= \frac{b}{(1+a)(1+b)} \\
d_1 &= \frac{(1+a)(1-b)+(1-a)(1+b)}{(1-a)(1+b)} \\
d_2 &= \frac{(1-a)(1-b)}{(1+a)(1+b)}
\end{aligned} \tag{13.5.13}$$

这就完成了带通滤波器的设计。

有时，设计者能够想出怎样对 $\mathscr{H}(f)$ 直接构造关于 $w$ 的有理函数，那就不必须从它的模平方开始进行设计。但为了稳定性，构造的函数只能在上半平面上设有极点。若用 $-w$ 代替 $w$，则函数还应该具有变成其自身复共轭的特性，所以滤波系数将是实型的。

例如，下面是槽形滤波函数，它被设计成仅仅去掉围绕某个可靠频率 $w=w_0$ 的窄频率带，其中 $w_0$ 是个正数，

$$\begin{aligned}
\mathscr{H}(f) &= \left(\frac{w-w_0}{w-w_0-i\varepsilon w_0}\right)\left(\frac{w+w_0}{w+w_0-i\varepsilon w_0}\right) \\
&= \frac{w^2-w_0^2}{(w-i\varepsilon w_0)^2-w_0^2}
\end{aligned} \tag{13.5.14}$$

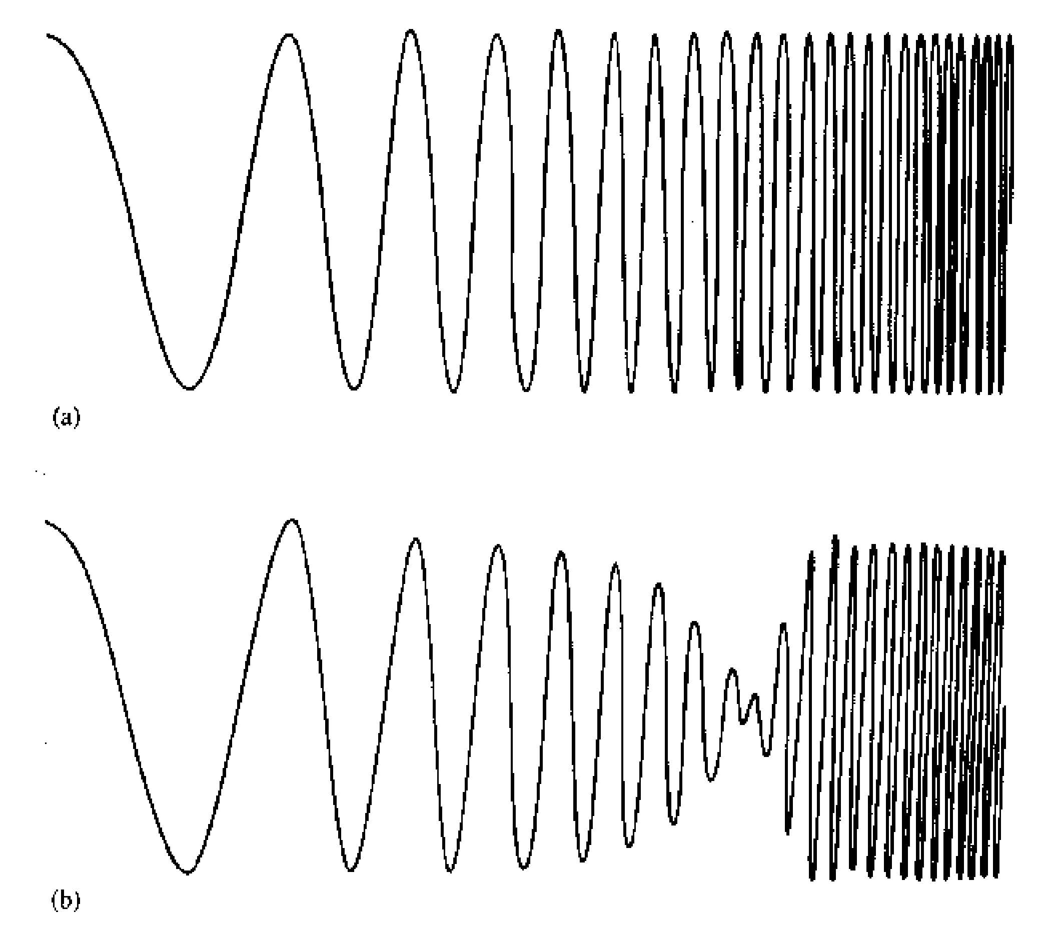
在式(13.5.14)中，参数 $\varepsilon$ 是一个小正数，是槽口的期望宽度，通过用 $z$ 代替 $w$ 的算术运算，给出滤波系数

$$\begin{aligned}
c_0 &= \frac{1+w_0^2}{(1+\varepsilon w_0)^2+w_0^2} \\
c_1 &= -2\frac{1-w_0^2}{(1-\varepsilon w_0)^2+w_0^2} \\
c_2 &= \frac{1+w_0^2}{(1+\varepsilon w_0)^2+w_0^2} \\
d_1 &= 2\frac{1-\varepsilon^2 w_0^2-w_0^2}{(1-\varepsilon w_0)^2+w_0^2} \\
d_2 &= -\frac{(1-\varepsilon w_0)^2+w_0^2}{(1+\varepsilon w_0)^2+w_0^2}
\end{aligned} \tag{13.5.15}$$

图13.5.1显示了对“鸟鸣”输入信号使用式(13.5.15)形式的滤波后所得出的结果。这种信号的频率向上渐变，一直跨过槽口频率。

双线性变换虽然看起来可能很一般，但它的应用由形成滤波的某些特征所限制。这种方法擅长于得到

所期望的滤波器的普通形式，适用于"平坦"的目标处。然而，$w$ 和 $f$ 之间的非线性映射，使得设计所期望的截止形状变得困难了，并且可能把截止频率（用定量的 db 定义）移离它们的期望位置。因而，数字滤波器的设计者们将双线性变换留用于特殊的场合，并且给它们本身配备大量的其它方法。我们建议读者按设计方案的要求同样来做一遍。



（a）一个"鸟鸣"信号，即频率随时间连续地增长。（b）通过槽形滤波器（13.5.15）后的同一信号。此处参数 $\epsilon$ 是 0.2。

图13.5.1

参考文献和进一步读物：


Hamming, R. W. 1983, *Digital Filters*, 2nd ed. (Englewood Cliffs, NJ: Prentice-Hall).

Antoniou, A. 1979, *Digital Filters: Analysis and Design* (New York: McGraw-Hill).

Parks, T. W., and Burus, C. S. 1987, *Digital Filter Design* (New York: Wiley).

Oppenheim, A. V., and Schafer, R. W. 1989, *Discrete-Time Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall).


## 13.6 线性预测和线性预测编码

我们从一个非常一般的公式出发，它能使我们与各种特殊情况联系起来。设 $\{y'_\alpha\}$ 是变量 $y$ 的一组测量数据，$y$ 的真实值表示为 $\{y_\alpha\}$，测量值和真实值之间有如下关系

$$y'_\alpha = y_\alpha + n_\alpha \tag{13.6.1}$$

（和等式(13.3.2)相比，可以看出符号上的一些差别）。我们采用希腊字母作为数据组的下

标，是为了表明数据点不是在一条直线上均匀分布或顺序排列的；例如，它们可以是三维空间中“随机”分布的点。现在，假设我们要对一些特殊的点 $y_*$ 的真实值进行最佳估计，$y_*$ 是已知的带有噪声的数据点的线性组合，即

$$y_* = \sum_\alpha d_{*\alpha} y'_\alpha + x_* \tag{13.6.2}$$

我们要找出函数 $d_{*\alpha}$，以使差值 $x_*$ 最小。系数 $d_{*\alpha}$ 有一个星号下标是为了表明它依赖于 $y_*$ 点的选择。后面，我们可能还要设 $y_*$ 做为 $y_\alpha$ 的一个存在值。在那种情况下，我们的问题成为一个最佳滤波或估计，和第13.3节中讨论的非常类似。而另一方面，我们可能希望 $y_*$ 成为全新的点，在这种情况下，我们的问题是一个线性预测的问题。

为使差值 $x_*$ 最小，我们自然从统计均方差考虑。如果用尖括号代表统计均方差，则我们所求的 $d_{*\alpha}$ 值是使下式达到最小的值：

$$\begin{aligned}\langle x_*^2\rangle &= \left\langle\left[\sum_\alpha d_{*\alpha}(y_\alpha + n_\alpha) - y_*\right]^2\right\rangle \\ &= \sum_{\alpha\beta}(\langle y_\alpha y_\beta\rangle + \langle n_\alpha n_\beta\rangle)d_{*\alpha}d_{*\beta} - 2\sum_\alpha\langle y_* y_\alpha\rangle d_{*\alpha} + \langle y_*^2\rangle\end{aligned} \tag{13.6.3}$$

其中，我们已经使用了噪声和信号不相关这个事实，即 $\langle n_\alpha y_\beta\rangle = 0$。量 $\langle y_\alpha y_\beta\rangle$ 和 $\langle y_* y_\alpha\rangle$ 描写了基本数据的自相关结构。在式(13.2.2)中对于一条直线上均匀分布的点，我们已经看到了类似的表达式。

在第14章和第15章我们还会几次遇到统计意义上的相关性。量 $\langle n_\alpha n_\beta\rangle$ 描写了噪声的自相关性质。通常对于点与点之不相关的噪声，存在关系式 $\langle n_\alpha n_\beta\rangle = \langle n_\alpha^2\rangle\delta_{\alpha\beta}$。于是我们可以方便地得到组成矩阵和向量的各种相关量。

$$\varphi_{\alpha\beta} \equiv \langle y_\alpha y_\beta\rangle \qquad \varphi_{*\alpha} \equiv \langle y_* y_\alpha\rangle \qquad \eta_{\alpha\beta} \equiv \langle n_\alpha n_\beta\rangle \quad 或 \langle n_\alpha^2\rangle\delta_{\alpha\beta} \tag{13.6.4}$$

将式(13.6.3)对 $d_{*\alpha}$ 微分并令其为 0，得到线性方程组：

$$\sum_\beta[\varphi_{\alpha\beta} + \eta_{\alpha\beta}]\,d_{*\beta} = \varphi_{*\alpha} \tag{13.6.5}$$

如果我们采用逆矩阵的形式，那么在忽略最小差值 $x_*$ 时估计式(13.6.2)可写成为：

$$y_* \approx \sum_{\alpha\beta}\varphi_{*\alpha}[\varphi_{\mu\nu} + \eta_{\mu\nu}]^{-1}_{\alpha\beta}\, y'_\beta \tag{13.6.6}$$

从式(13.6.3)和(13.6.5)我们还可以计算出差值 $x_*$ 的最小均方值，记为 $\langle x_*^2\rangle_0$：

$$\langle x_*^2\rangle_0 = \langle y_*^2\rangle - \sum_\beta d_{*\beta}\varphi_{*\beta} = \langle y_*^2\rangle - \sum_{\alpha\beta}\varphi_{*\alpha}[\varphi_{\mu\nu} + \eta_{\mu\nu}]^{-1}_{\alpha\beta}\,\varphi_{*\beta} \tag{13.6.7}$$

最后，我们看一个一般的结果，它告诉我们如果在式(13.6.2)中不用最佳值 $d_{*\beta}$，而用一些别的值 $\hat{d}_{*\beta}$，看均方差值 $\langle d_{*\beta}\rangle$ 将增加多少。上面的等式隐含着：

$$\langle x_*^2\rangle = \langle x_*^2\rangle_0 + \sum_{\alpha\beta}(\hat{d}_{*\alpha} - d_{*\alpha})[\varphi_{\alpha\beta} + \eta_{\alpha\beta}](\hat{d}_{*\beta} - d_{*\beta}) \tag{13.6.8}$$

因为第二项是一个四次式，所以我们可以看到差值的增加仅仅是估计 $d_{*\beta}$ 值时产生的误差的二次式。

### 13.6.1 和最佳滤波的联系

如果我们将“星号”变为希腊下标，比如 $\gamma$，则上面的等式描述的正是最佳滤波，它比第13.3节中讨论的更为普遍。例如，如果噪声 $n_\alpha$ 的幅度变为零，那么噪声自相关 $\eta_{\alpha\beta}$ 也将变为零，矩阵和它的逆矩阵相乘将互相抵消，于是，式(13.6.6)就简化成为 $y_\gamma = y'_\gamma$。还有一种特

殊情况是，矩阵 $\varphi_{\alpha\beta}$ 和 $\eta_{\alpha\beta}$ 是对角矩阵，在这种情况下，方程(13.6.6)成为：

$$\hat{y}_\gamma = \frac{\varphi_{\gamma\gamma}}{\varphi_{\gamma\gamma} + \eta_{\gamma\gamma}} y'_\gamma \tag{13.6.9}$$

和等式(13.3.6)相比较可以看出，$S^2 \to \varphi_{\gamma\gamma}$，$N^2 \to \eta_{\gamma\gamma}$。实际情况是：对于均匀间隔数据点的情况，在傅里叶变换域中，自相关就很简单地成为傅里叶振幅的平方(Wiener—Khimchin 定理，见等式(12.0.12))，最佳滤波可以从式(13.6.9)构造，而不需要对矩阵求逆。

更一般的情况，在时域或其它域中，最佳滤波(在测量噪声存在时，从基本的真实值中求差值平方极小化)可以通过估计自相关矩阵 $\varphi_{\alpha\beta}$ 和 $\eta_{\alpha\beta}$ 以及运用式(13.6.6)(将 $* \to \gamma$)来构造。(事实上，式(13.6.8)是第13.3节中叙述情况的基础，在那里即使是粗糙的最佳滤波也能十分有效。)

## 13.6.2 线性预测

经典的线性预测讨论的是一种特殊情况，即数据点 $y_\beta$ 在一条线上等间隔分布 $y_i(i=1,2,\ldots,N)$，并且我们要求用 $M$ 个连续点 $y_i$ 来预测第 $M+1$ 点。我们假设其是平稳的，即 $\langle y_j y_k \rangle$ 的自相关取决于差值 $|j-k|$，而不单独地依赖于 $j$ 和 $k$，因此自相关 $\varphi$ 只有一个下标：

$$\varphi_j \equiv \langle y_i y_{i+j} \rangle \approx \frac{1}{N-j} \sum_{i=1}^{N-j} y_i y_{i+j} \tag{13.6.10}$$

其中近似等式说明了如何用实际的数据组来估计自相关分量(实际上，还有一个更好的方法进行这种估计，参看下式)。在我们所描述的情况下，式(13.6.2)的估计式是

$$y_n = \sum_{j=1}^{M} d_j y_{n-j} + x_n \tag{13.6.11}$$

(和式(13.5.1)相比)等式(13.6.5)成为含 $M$ 个未知数 $d_j$ 的 $M$ 个方程的方程组，$d_j$ 称为线性预测系数(*LP* 系数)

$$\sum_{j=1}^{M} \varphi_{|j-k|} d_j = \varphi_k \quad (k = 1, \ldots, M) \tag{13.6.12}$$

注意，尽管噪声没有明显地包含在上面的等式中，但如果点与点不相关，它还是被正确地考虑进去了：根据式(13.6.10)用测量值 $y'_i$ 估计的 $\varphi_0$，实际估计了上面的对角部分 $\varphi_{\alpha\alpha} + \eta_{\alpha\alpha}$。根据等式(13.6.7)可以估计均方差值 $\langle x_n^2 \rangle$ 为：

$$\langle x_n^2 \rangle = \varphi_0 - \varphi_1 d_1 - \varphi_2 d_2 - \ldots - \varphi_M d_M \tag{13.6.13}$$

为了运用线性预测，我们首先利用式(13.6.10)和(13.6.12)计算 $d_j$，接着计算等式(13.6.13)。具体说来，就是将式(13.6.11)用于已知的记录数据中而求得差值 $x_i$ 有多大。如果差值小，则我们继续应用等式(13.6.11)进行下一步的运算，估计下一步的差值 $x_i$，直到差值为0。在这种运用情况下，式(13.6.11)是一种外推表达式。在很多情况下，可以发现这种外推比任何一种简单的多项式外推更加强有力。(顺便提一句，请不要混淆了"线性预测"和"线性外推"这两个概念；线性预测的一般函数表达式比直线甚至低阶的多项式复杂得多。)

但是，为了充分利用线性预测，必须注意它所限制的一面：我们必须进行额外的测量以保证它的**稳定性**。等式(13.6.11)是一般的线性滤波式(13.5.1)的特殊情况。等式(13.6.11)作为线性预测器能够稳定的条件已精确地由等式(13.5.5)和(13.5.6)表达了，也就是特征多项式：

$$z^N - \sum_{j=1}^{N} d_j z^{N-j} = 0 \qquad (13.6.14)$$

它使 $N$ 个根都落在单位圆内，即

$$|z| \leq 1 \qquad (13.6.15)$$

我们不能保证由等式(13.6.12)确定的系数有这种性质。如果数据点包含许多振荡，而这些振荡没有任何振幅增加或减少的特定趋势，则式(13.6.14)中的复根将都和单位圆靠得很近。数据组的有限长度也将使一些根在单位圆内，而另一些根在单位圆外。在某些应用中，结果的不稳定性是缓慢上升的，并且线性预测不会遍离得太远，因此最好直接用从式(13.6.12)导出的"没有加工"过的线性预测系数。例如，如果可以将一组数据外推一小步；那么可以从数据出发按此步长分别向前和向后外推。如果两个外推符合得非常好，则不稳定性便不是一个问题。

当不稳定性成为问题时，就必须对线性预测系数进行"加工处理"，可以按如下步骤进行：(i)解等式(13.6.14)得到 $N$ 个复根(用数字解法)；(ii)将根移到认为应该是在单位圆内或在单位圆上的地方；(iii)重新构造现在已经修正过的 LP 系数。读者可能会认为第(ii)步有一点含糊。确实如此。没有更好的过程了。如果认为信号确实是一组不衰减的正弦和余弦波之和(可能具有不相称的周期)，那么只需简单地将每个根 $z_i$ 移到单位圆上：

$$z_i \to z_i / |z_i| \qquad (13.6.16)$$

在另外的情况下，将一些比较糟糕的根映射到圆内可能比较合适

$$z_i \to 1/z_i^* \qquad (13.6.17)$$

这种选择的替换方法有一个性质，即当式(13.6.11)输出的振幅由一组 $x_i$ 的正弦曲线驱动时，其振幅维持不变。这里我们假设式(13.6.12)能够正确地鉴别共振的谱宽度，但是仅疏忽了鉴别它的时间含意，以使应随时间衰减的信号按振幅增长而结束。对式(13.6.16)和(13.6.17)的选择有一点随意性，但我们倾向于式(13.6.17)。

还有一个问题是 $M$ 的选择，即使用的线性预测系数。应该根据工作规模选择 $M$ 尽可能小，也就是说，必须由实验来确定所选择的数据点。可以试取 $M=5,10,20,40$。如果需要的 $M$ 比这个值更大，要注意那些复根的"处理"过程对于舍入误差非常敏感。所以请采有双精度。

线性预测对于光滑和振荡信号的外推特别有用，尽管不要求信号是周期性的。在这种情况下，通过信号的很多次循环线性预测通常外推得非常精确。相比之下，多项式外推一般在经过一两次循环之后就变得很不精确。可以成功地进行线性预测的信号原型事例是，海洋潮汐的高度，对于此信号，基本的 12 小时周期信号的相位和振幅随着年月进行调整，当地的水压效应会使一个周期的曲线的形状和正弦波相当不同。

我们已经注意到，从数据集来估算协方差 $\varphi_k$，等式(13.6.10)并不是最好途径。实际上，线性预测所得结果对 $\varphi_k$ 值估计的精确程度非常敏感。有一个特别好的方法，此方法应归功于伯格(Burg)[1]，它是一个迭代过程，每次迭代 $M$ 的阶数加 1。在每一步迭代中重新估计系数 $d_j(j=1,\ldots,M)$ 以便使式(13.6.13)中的残差尽可小的。尽管伯格方法的进一步讨论超出了我们的范围，但下面的程序[1,2]给出了从一组数据估计线性预测系数 $d_j$ 的方法。

```
#include <math.h>
#include "nrutil.h"
```

```
void memcof(float data[], int n, int m, float *xms, float d[])
    已知实型向量data[1..n]和m,此程序返回m个线性预测系数为d[1..m],同时均方差值以xms返回。
{
    int k,j,i;
    float p=0.0, *wk1, *wk2, *wkm;

    wk1=vector(1,n);
    wk2=vector(1,n);
    wkm=vector(1,m);
    for (j=1;j<=n;j++)  p += SQR(data[j]);
    *xms=p/n;
    wk1[1]=data[1];
    wk2[n-1]=data[n];
    for (j=2;j<=n-1;j++) {
        wk1[j]=data[j];
        wk2[j-1]=data[j];
    }
    for  (k=1;k<=m;k++) {
        float num=0.0,denom=0.0;
        for (j=1;j<=(n-k);j++) {
            num += wk1[j]*wk2[j];
            denom += SQR(wk1[j])+SQR(wk2[j]);
        }
        d[k]=2.0*num/denom;
        *xms *= (1.0-SQR(d[k]));
        for (i=1;i<=(k-1);i++)
            d[i]=wkm[i]-d[k]*wkm[k-i];     这种算法是迭代的,随m值越来越大计算迭代的结果,直到
                                           取得所要求的值。在此算法中,此处程序返回向量d和标量
                                           xms,它们是由k项(而不是m项)组成的一组LP系数。
        if (k == m) {
            free_vector(wkm,1,m);
            free_vector(wk2,1,n);
            free_vector(wk1,1,n);
            return;
        }
        for  (i=1;i<=k;i++) whm[i]=d[i];
        for  (j=1;j<=(n-k-1);j++) {
            wk1[j] -= wkm[k]*wk2[j];
            wk2[j]=wk2[j+1]-wkm[k]*wk1[j+1];
        }
    }
    nrerror("never get here in memcof.");
}
```

下面是使LP系数变得稳定的过程(如果需要这样做的话),用原始的或加工过的LP系数对一组数据进行线性预测。程序**zroots**(第9.5节)用来求一个多项式的所有复根。

```
#include <math.h>
#include "complex.h"
#define NMAX 100                        m为最大可能值
#define ZERO Complex(0.0,0.0)
#define ONE Complex(1.0,0.0)

void fixrts(float d[],int m)
    对于给定的LP系数d[1..m],此程序找出特征多项式(13.6.14)的所有根,将单位圆外的根映射到单位圆内,接着
    返回处理过的修正系数d[1..m]
{
    viod zroots(fcomplex a[], int m, fcomplex roots[], int polish);
```

```
    int i,j,polish;
    fcomplex a[NMAX],roots[NMAX];

    a[m]=ONE;
    for (j=m-1;j>=0;j--)
        a[j]=Complex(-d[m-j],0.0);          设置所求根的多项式系数为复数值
    polish=1;
    zroots(a,m,roots,polish);               求出所有的根
    for (j=1;j<=m;j++)                      寻找单位圆外的根并将它映射到圆内
        if (Cabs(roots[j]) > 1.0)
            roots[j]=Cdiv(ONE,Conjg(roots[j]));
    a[0]=Csub(ZERO,roots[1]);               新构成多项式系数
    a[1]=ONE;
    for (j=2;j<=m;j++) {                    通过对根循环和综合乘法重
        a[j]=ONE;
        for (i=j;i>=2;i--)
            a[i-1]=Csub(a[i-2],Cmul(roots[j],a[i-1]));
        a[0]=Csub(ZERO,Cmul(roots[j],a[0]));
    }
    for (j=0;j<=m-1;j++)                    多项式的系数是实数所以我们
        d[m-j] = -a[j].r;                   只需退回实部做为新的 LP 系数
    }

include "nrutil.h"

void predic(float data[], int ndata, float d[], int m, float future[], int nfut)
    已知数据 data[1..ndata]和数据的 LP 系数 d[1..m],此程序运用等式(13.6.11)预测后面 nfut 个数据点,这些数据
    点以数组 future[1..nfut]返回。注意程序只用 data 数组最后 m 个数据点作为预测的初始参考值。
{
    int k,j;
    float sum,discrp, *reg;

    include "nurtil.h"

    void predic(float data[], int ndat, float d[], int m, float future[], int nfut)

    reg=vector(1,m);
    for (j=1;j<=m;j++) reg[j]=data[ndata+1-j];
    for (j=1;j<=nfut;j++) {
        discrp=0.0;                 如果是通过线性预测程序重新构造一个函数而不是通过线性预测
                                    外推一个函数,那么在此应该输入已知的差值
        sum=discrp;
        for (k=1;k<=m;k++) sum += d[k]*reg[k];
        for (k=m;k>=2;k--) reg[k]=reg[k-1];        [若要履行循环数组
        future[j]=reg[1]=sum;                      可以免去系数的移位]
    }
    free_vector(reg,1,m);
}
```

### 13.6.3 除掉线性预测的偏差

人们可能会期望等式(13.6.11)中 $d_j$ 之和(或者更普遍一些,在式(13.6.2)中)应该是 1,以便使对所有的数据点 $y_i$ 增加一个常数后所产生的预测也增加一个相同的常数。但是 $d_j$ 之和并不是 1 ,通常比 1 稍小。这个事实揭示了非常微妙的一点,那就是经典的线性预测不是无偏的,尽管它使均方差值最小。在被测的自相关不是较佳估计值的地方,线性预测方程组倾向于预测趋向零的值。

有时，那正是人们所需的。如果产生 $y_i$ 的过程确实有零平均值，则在缺少其它信息的情况下，零将是最佳估值。然而，在别的情况下，这是没有根据的。如果数据显示出围绕其正值有微小的变化，便不需要趋向零的线性预测。

通常还有一个非常有效的近似，那就是从数据组中减去平均值，再进行线性预测，接着再把平均值加上。这一过程包含了正确答案的萌芽，但简单的算术平均值并不是要减去的正确的常数。实际上，无偏估计式是通过对各点减去由下式定义的自相关加权平均值而得到的[3,6]：

$$\overline{y} \equiv \sum_{\beta}[\varphi_{\mu\alpha}+\eta_{\mu\alpha}]^{-1}_{\alpha\beta}\, y_\beta / \sum_{\alpha\beta}[\varphi_{\mu\alpha}+\eta_{\mu\alpha}]^{-1}_{\alpha\beta} \tag{13.6.18}$$

通过这种减法后，LP 系数之和应该是单位量。不论 $\varphi_k$ 是怎样估计，它都符合舍入误差和差值。

### 13.6.4 线性预测编码(LPC)

一个不同的，但是有关联的方法，即应用上面的等式对采样信号进行“压缩”，以便使信号更紧凑地储存起来。原始的信号又应该能从已压缩形式中精确地恢复出来。显然，仅当信号有冗余度时压缩才能实现。等式(13.6.11)描述了一种冗余度：它说明除了一点小的偏差外，一个信号可以从它前面的值以及少量的 LP 系数预测出来。因此利用式(13.6.11)进行信号的压缩称为**线性预测编码**，或 LPC。

LPC 最基本的思想(用最简单的形式)是记录下述数据作为一个压缩的文件：(i)LPC 系数的数目 $M$；(ii)由子程序 **memcof** 得到的 $M$ 个值；(iii)开始的 $M$ 个数据点以及(iv)以后每个点只要求它的残差 $x_i$(等式(13.6.1))。当正在生成压缩文件时，运用等式(13.6.1)于前面的 $M$ 个点，并从当前点的实际值减去和式而求得残差。当重新恢复原文件时，将残差所加到程序 **predic** 所显示的各点上。

可能读者还不是很明白为什么要用这种方法进行数据压缩。对每个数据点我们是存储残差值！为什么不存储原始数据值呢？答案取决于我们所涉及的数目的相对大小。残差是通过两个非常相近的数(数据和线性预测值)相减而得到的，因此，通常差值的非零比特位数目非常小。这样，这些数据就能存储在一个被压缩的文件中。那么，怎样用高级语言做到这些呢？下面是一种方法：改变数据的尺寸，使它们成为整数值，比如在 +1000000 和 −1000000 之间(假设需要六位有效数字)。调整等式(13.6.1)，使之包含上面“整数部分”操作的和。现在通过定义，差值将是整数值。对不同的 $M$ 值实验，寻找 LP 系数使差值尽可能小，小到允许的程度。如果能达到 ±127 的范围(在我们的实验中，这一点并不难)，那么，就可以在文件中用一个字节将差值写下来。和原来数据用 4 个字节的整数和浮点数相比，该压缩因子为 4 。

注意 LP 系数是用量化的数据计算的，差值也是量化了的，也就是说，在 LPC 循环的内外都是量化的数值。如果仔细地按上面的步骤进行，那么除了最初对数据量化外，在压缩重建过程中不会引入任何，哪怕是单个比特位的舍入误差。尽管在对式(13.6.11)求和时会引入舍入误差，但所存储的残差在加回求和式时，给出了精确的原始(量化)数据值。还请注意，不必为了稳定性对 LP 系数进行加工处理；通过对各个点加上残差，将永远不会偏离原始数据，所以不会增加不稳定性。因此，没有必要象上面那样运行程序 **fixrts**。

在第20.4节中将学习**霍夫曼**(Huffman)**编码**,该方法将利用小的差值出现频率大于大的差值出现频率这一事实,进一步压缩残差。最初的霍夫曼编码采用下面形式,如果大多数的差值在 ±127 的范围内,只是偶尔一两个差值超出这范围,那么用数据值 127 表示为"超出范围",接着对超出范围的差值之全字值记录于文件中(紧随 127 之后)。第20.4节将解释怎样能做得更好。

有许多不同的处理过程都属于 LPC 的范畴。

- 如果数据的特征是时间的变量,则最好不要用整组数据来求一组 LP 系数,而是将数据分成几段,对每一段分别计算和存储不同的 LP 系数。
- 如果数据确能用它们的 LP 系数很好地表征的话,而且又能够容忍小量的误差,则不必麻烦去存储所有的残差。可以直接进行线性预测直到误差超出容差限的范围,然后再重新初始化(用 $M$ 个顺序存储的残差),并继续预测。
- 在某些应用情况下,最明显的应用是语言合成,人们关心的仅仅是重建信号的谱幅度,而不是相对相位。在这种情况下,人们不需要存储任何初始数据,仅仅是各个数据段的 LP 系数。输出的重建是通过这些系数和所有数据都为 0 的初始条件来完成的,除了一个非零的尖峰外。一个语言合成芯片可有多达 10 阶的 LP 系数,它每秒可能变化 20 到 50 次。
- 有些人相信即使残差 $x_i$ 不是很小,用 LPC 分析信号也非常有趣。此时 $x_i$ 被解释为基本的"输入信号",当通过由 LP 系数定义的多极点滤波器滤波后,它成为被观察的"输出信号"。据说,LPC 同时揭示了滤波器的本质以及驱动它的特定输入信号的性质。我们对这种应用表示怀疑,这种文献过于夸张了。

参考文献和进一步读物:


Childers, D. G. (ed) 1978, *Modern Spectrum Analysis* (New York: IEEE Press), especially the paper by J. Makhoul(reprinted from *Proceedings of the IEEE*, vol. 63, p. 561, 1975).

Burg, J. P. 1968, reprinted in Childers 1978. [1]

Anderson, N. 1974, reprinted in Childers, 1978. [2]

Cressie, N. 1991, *in Spatial Statistics and Digital Image Analysis*(Washington: National Academy Press). [3]

Press, W. H., and Rybicki, G. B. 1992, *Astrophysical Journal*, vol. 398, (Oct, 10. 1992). [4]


## 13.7 用最大熵(全极)方法的功率谱估计

FFT 并不是一个过程的功率谱估计的唯一途径,也不一定是达到目的最好的方法。为了弄清怎样构想另外一种方法,我们首先扩大一下视野,使之不仅仅包括奈奎斯特(Nyquist)区间 $-f_c<f<f_c$ 内的实数频率,而且包括整个复频率平面。从这一角度出发,让我们通过下式将复 $f$ 平面变换一个新的平面,称为 **z 变换平面或 z 平面**

$$z \equiv e^{2\pi i f\Delta} \tag{13.7.1}$$

其中 Δ 和通常情况一样,是时域的取样间隔。注意 $f$ 平面上实轴奈奎斯特区间和复 $z$ 平面上的单位圆有一一对应关系。

如果我们将式(13.7.1)与(13.4.4)和(13.4.6)比较,则可以看出,在不考虑归一化条件时,对任何实

取样函数 $c_k = c(t_k)$ 的 FFT 功率谱估计可写为

$$P(f) = \left| \sum_{k=-N/2}^{N/2-1} c_k z^k \right|^2 \tag{13.7.2}$$

当然式(13.7.2)不是基本函数 $c(t)$ 的真正功率谱，而仅仅是个估计。从下面两个有关方面可以看出为什么这个估计值不是精确的。首先在时域里，估计值仅仅是以函数 $c(t)$ 有限的一段为基础，而实际上我们知道 $c(t)$ 从 $t=-\infty$ 到 $\infty$ 是连续的。其次，在式(13.7.2)的 $z$ 平面中，通常情况，有限的洛伦特(Laurent)级数仅仅对 $z$ 的一般解析函数提供一个近似值。事实上，一个表示"真正"功率谱的正规表达式(归一化)是

$$P(f) = \left| \sum_{k=-\infty}^{\infty} c_k z^k \right|^2 \tag{13.7.3}$$

这是一个洛伦特(Laurent)无穷级数，依赖于无穷个 $c_k$ 值。等式(13.7.2)仅仅是由式(13.7.3)确定的 $z$ 的解析函数的一种解析近似。事实上，这种近似隐含在用 FFT 以周期图方法进行功率谱估计中。它有几种命名方法，包括**直接方法**、**全零点模型**和**移动平均($MA$)模型**。"全零点"是特指模型的谱在 $z$ 平面上有零点，但没有极点。

如果我们从更广的角度看式(13.7.3)的近似问题，就可以发现如果采用一个有理函数，在其分子和分母都采用式(13.7.2)型的级数，我们就可能做得更好。还有一个表达式，尽管不是很明显，但在近似时却有一些有利之处，这个近似值的自由参数都在分母上，也就是

$$P(f) \approx \frac{1}{\left| \sum_{k=-M/2}^{M/2} b_k z^k \right|^2} = \frac{a_0}{\left| 1 + \sum_{k=1}^{M} a_k z^k \right|^2} \tag{13.7.4}$$

在这里，第二个等号引入了一组新的系数 $a_k$，它可以根据 $z$ 都在单位圆上由 $b_k$ 求得。而 $b_k$ 可以根据式(13.7.4)的幂级数展开应和式(13.7.3)前面 $M+1$ 项一致的条件来确定。实际上，我们还有其它的方法确定 $b_k$ 或 $a_k$。

近似式(13.7.2)和(13.7.4)的区别不仅仅是形式上的。它们的近似也具有不同的特征。最明显的是式(13.7.4)可以有**极点**，它对应于 $z$ 单位圆上，也就是奈奎斯特区间的实频率上，无穷大的功率谱密度。这种极点精确地代表了具有离散的、陡峭的"直线"或 $\delta$ 函数的基本功率谱。相比之下，式(13.7.2)在奈奎斯特区间的实频率上只有零点，而没有极点。因此必须用多项式才能适合具有尖峰的谱的特征。式(13.7.4)近似式有几个名字：**全极点模型**、**最大熵方法(MEM)**或**自回归模型(AR)**。我们只需知道怎样从数据组计算系数 $a_0$ 和 $a_k$，就能用式(13.7.4)获得谱估计。

令人们惊讶而高兴的是，我们已经知道怎样计算！回到前面看看线性预测的式(13.6.11)。将它和线性滤波等式(13.5.1)和(13.5.2)相比较，就可以发现从滤波器的角度将 $x$ 作为输入信号，$y$ 作为输出信号，线性预测有一个滤波函数

$$\mathscr{H}(f) = \frac{1}{1 - \sum_{j=1}^{N} d_j z^j} \tag{13.7.5}$$

因此，$y$ 的功率谱等于 $x$ 的功率谱乘以 $|\mathscr{H}(f)|^2$。现在，让我们考虑当 $x$ 是线性预测的残差时，这个输入信号 $x$ 的功率谱是什么。虽然我们不作正式证明，但从直观上我们可以相信，由于 $x$ 是独立随机的，因此具有平坦的谱(白噪声)。(粗略地说，$x$ 中遗留的任何残差关系容许一个更精确的线性预测，并且这种残差关系也可以被删除)。这种平坦谱的整体归一化就是 x 的均方振幅。这正是由式(13.6.13)计算的精确量，并且通过程序 **memcof** 以 xms 返回。因此等式(13.7.4)中的系数 $a_0$ 和 $a_k$ 与由 **memcof** 返回的 LP 系数存在下面的简单关系

$$a_0 = \text{xms} \quad a_k = -\text{d}(k), \quad k = 1, \ldots, M \tag{13.7.6}$$

还有另外一种方法描述 $a_k$ 和自相关分量 $\varphi_k$ 的关系，Winener-Khinchin 公理(12.0.12)说明自相关的傅里叶变换等于功率谱。用 $z$ 变换的语言来说，这种傅里叶变换就是用 $z$ 表示的洛伦特(Laurent)级数，确

定系数的等式(13.7.4)就成为

$$\frac{a_0}{\left|1+\sum_{k=1}^{M} a_k z^k\right|^2} \approx \sum_{j=-M}^{M} \varphi_j z^j \tag{13.7.7}$$

等式(13.7.7)中的近似号具有一种特殊的解释。它意味着，左边幂级数展开式和右边的各项从 $z^{-m}$ 到 $z^m$ 都是一致的。在此范围外，右边各项均为 0，而左边还有非零项。注意，$M$ 近似等式左边的系数的个数，它可以最大到 $N$ 的任一整数，$N$ 是可用的自相关所有数目(实际中，$M$ 可以选择得比 $N$ 小得多)。$M$ 被称为逼近的阶数或极点的个数。

对任何选择的 $M$ 值，式(13.7.7)左边的级数展开定义了滞后大于 $M$ 的自相关函数的某种外推，事实上甚至可以超出 $N$，也就是说超出了实际所测量的数据范围。从信息论角度，可以证明在所有可能的外推中，这种外推具有最大熵。因此称为**最大熵方法**或 MEM。MEM 方法的最大熵性质使它获得让人盲从的流行性。有人说它本身给出一个比其它方法"更好"的估计，不要相信这种说法。MEM 确实有让人信服的能够用于拟合带有陡峭的谱特征的性质，但对于功率谱估计它并没有其它任何魔力。

在程序 **memcof** 中操作数标尺为 $N$(数据点的个数)和 $M$ 的乘积(MEM 近似期待的阶数)。如果 $M$ 选择得和 $N$ 一样，那么 MEM 方法比前一节 $N\log N$ 量级的 FFT 慢得多。然而实际上，人们通常限制 $MEM$ 的阶数(或极点个数)近似为几倍于要估计的谱所具有的陡峭谱线的个数。由于对极点个数进行了这种限制，这种方法能在一定程度上使谱平滑，但这通常是人们希望得到的性质。尽管精确值决定于实际应用，人们通常取 $M=10$ 或 20 或 50 而 $N=1000$ 或 10000。在这种情况下，EME 估计不比 FFT 估计慢多少。

我们觉得有必要提醒读者程序 memcof 有时有些古怪。如果极点的个数或数据的数目太大了，即使采用双精度，舍入误差也会成为一个问题。对于"尖峰"数据(也就是具有非常尖的谱线特征数据)，甚至在中等大小阶数时，这种算法可能会将尖峰分开，并且尖峰还会随正弦波的相位而移动。另外，对于带有噪声的输入函数。如果选择的阶数太高，将会发现很多假尖峰！一些专家建议将这种算法和一些保守的算法如周期图法结合起来使用，以帮助选择正确的模型阶数，避免被假的谱线特征所迷惑。MEM 方法有时有点古怪，但是它也能做非常突出的工作。我们建议读者对你自己的问题仔细地试用它。现在我们从对系数的讨论转到对 MEM 谱估计的计算。

MEM 估计式(13.7.4)是连续变化频率 $f$ 的函数。如同 FFT 那样，对频率均匀划分没有任何特别的意义。事实上，因为 MEM 估计可能有非常尖的谱线特征，人们希望在这些特征线附近用非常精细的步骤来求值，而在远离这些特征线的地方粗糙一些。下面的函数，在系数已经计算出来后，估计等式(13.7.4)，并且以 $f\Delta$ 为变量的函数形式返回估计的功率谱。$f\Delta$ 应位于奈奎斯特区间 $-1/2\sim1/2$。

```
#inclued <math.h>

float evlmem(float fdt, float d[], int m, float xms)
    利用 memcof 返回已知的 d[1..m],m,xms,此函数以 fdt=fΔ 为变量,返回功率谱 P(f)。
{
    int i;
    float sumr=1.0,sumi=0.0;
    double wr=1.0,wi=0.0,wpr,wpi,wtemp,theta;          置循环量为双精度型

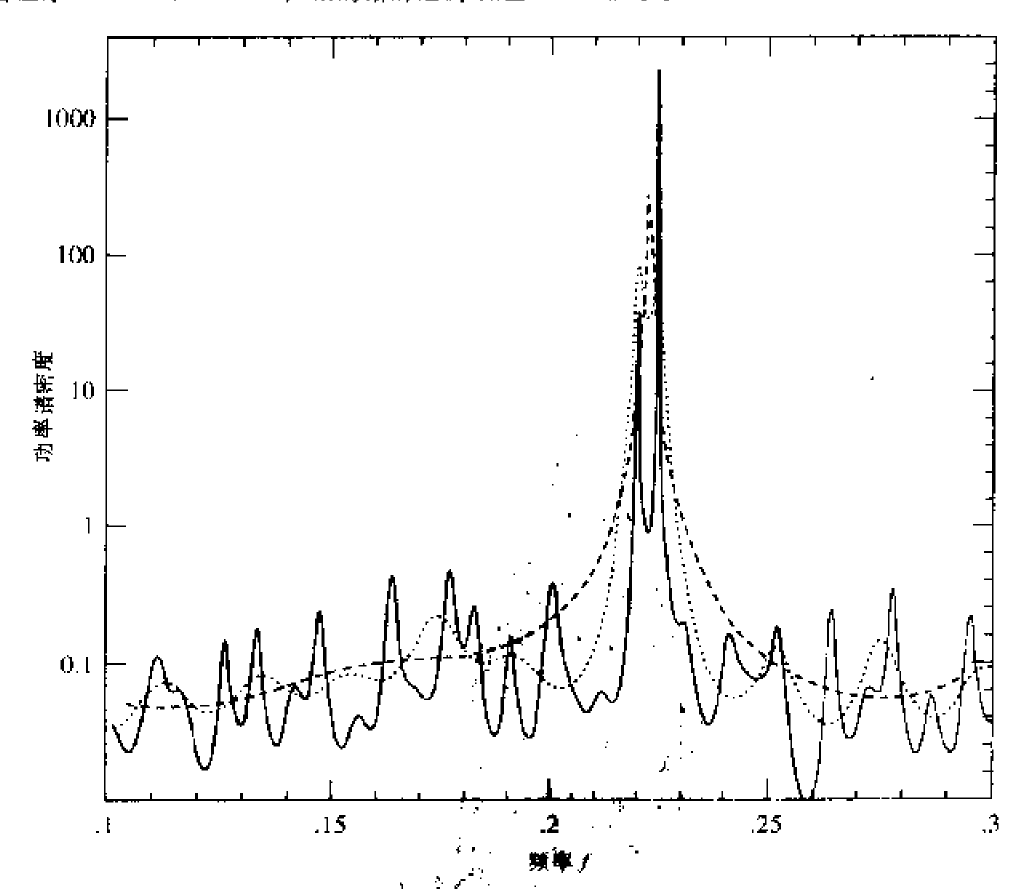
    theta=6.28318530717959*fdt;
    wpr=cos(theta);                                    建立循环关系
    wpi=sin(theta);
    for (i=1;i<=m;i++){                                对求和中的各项循环
        wr=(wtemp=wr)*wpr-wi*wpi;
        wi=wi*wpr+wtemp*wpi;
        sumr -= d[i]*wr;                               此处的累积用于计算式(13.7.4)的分母
        sumr -= d[i]*wr;
    }
    return xms/(sumr*sumr+sumi*sumi);                  等式(13.7.4)
```

}

注意要确保用非常精细的网格估计 $P(f)$，以便寻找非常窄峰。这种窄的峰如果出现，可能会包括数据中的所有功率。用户可能希望知道由程序 **memcof** 和 **evlmem** 产生 $P(f)$ 是怎样对输入数据向量到均方值归一化的。答案是：

$$\int_{-\frac{1}{2}}^{\frac{1}{2}} P(f\Delta)d(f\Delta) = 2\int_{0}^{\frac{1}{2}} P(f\Delta)d(f\Delta) = \text{数据的均方值} \tag{13.7.8}$$

由程序 **memcof** 和 **evlmen** 产生的谱的范例 如图13.7.1所示。




输入信号由频率非常接近的两个正弦信号之和的512个采样点组成，再加上约为均匀功率谱的白噪声。图示的是整个奈奎斯特频率区间的扩展部分（它从0延伸到0.5）。图中短划线表示的谱估计使用了20个极点；点线表示的谱估计使用了40个极点；这种方法可以明显地区分两个正弦信号，但是白噪声背景也开始显出假峰（注意图中的对数刻度）。

图13.7.1　用最大熵谱估计的输出范例

## 13.8　非均匀取样数据的谱分析

迄今为止，我们一律都是处理均匀取样的数据点

$$h_n = h(n\Delta) \qquad n = \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots \tag{13.8.1}$$

其中的 $\Delta$ 是取样间隔，它的倒数是取样率。回忆第12.1节由取样定理说明的奈奎斯特临界频率

$$f_c = \frac{1}{2\Delta} \tag{13.8.2}$$

其意义是：由等式(13.8.1)取样的数据集包含了信号 $h(t)$ 在奈奎斯特频率内所有谱分量的全部信息，而且

混迭或杂混了频率大于奈奎斯特频率的任何信号分量的信息。因此取样定理说明了对均匀取样数据分析时的有利和不利之处。

但在有些情况下得不到均匀间隔的数据。一个普遍的情况是仪器的数据丢失,因此获得的数据只是等式(13.8.1)的子集(非连续的整数),即所谓数据丢失的问题。还有一个普遍的情况存在于观察科学例如天文学,此时观察者不能完全控制观察时间,而必须接受某个受限制的时间集 $t_i$。

从非均匀间隔的 $t_i$ 到等式(13.8.1)所示的均匀间隔,有某些显而易见的方法。插值是一种方法:在数据上设置一个时间均匀间隔的网,在网结点上插值然后用 FFT 方法。对丢失数据的问题,只需插入丢失的数据点。如果很多连续数据点丢失了,最好令它们为 0,或者强制放到最后测量的点。但是,实践者的经验表明这种插入技巧并不让人放心。一般说来,这种技巧的效果很差。例如,数据的大步跳跃会导致功率谱在低频成分处(波长和步长相比)假的凸起。

对非均匀取样数据谱分析的一个完全不同的方法,能够克服上面的困难,并且具有一些其它的让人满意的性质,它是由罗姆(Lomb)[1]发展起来的,他的工作是建立在早期巴宁(Barning)[2]和范尼提(Vanicek)[3]工作的基础上的,并由斯卡格尔(Scargle)[4]进一步完善。罗姆方法(我们以后都这样称呼)仅估计在实际测量的时刻 $t_i$ 的数据,正弦的或余弦的。假设有 $N$ 个数据点 $h_i=h(t_i)(i=1,\ldots,N)$,则首先通过一般的关系求均值和方差

$$\bar{h} \equiv \frac{1}{N}\sum_{1}^{N} h_i \qquad \sigma^2 \equiv \frac{1}{N-1}\sum_{1}^{N}(h_i-\bar{h})^2 \tag{13.8.3}$$

现在,按下式定义**罗姆归一化周期图**(功率谱作为角频率 $\omega\equiv 2\pi f>0$ 的函数)

$$P_N(\omega) \equiv \frac{1}{2\sigma^2}\left\{ \frac{\left[\sum_j (h_j-\bar{h})\cos\omega(t_j-\tau)\right]^2}{\sum_j \cos^2\omega(t_j-\tau)} + \frac{\left[\sum_j (h_j-\bar{h})\sin\omega(t_j-\tau)\right]^2}{\sum_j \sin^2\omega(t_j-\tau)} \right\} \tag{13.8.4}$$

这里,$\tau$ 有关系式

$$\tan(2\omega\tau) = \frac{\sum_j \sin 2\omega t_j}{\sum_j \cos 2\omega t_j} \tag{13.8.5}$$

常数 $\tau$ 是一种偏移量,它使得 $P_N(\omega)$ 在所有 $t_i$ 移动一个常数时不变。罗姆还说明这种偏移量的选择还有一个更深远的效应:它使等式(13.8.4)和通过最小二乘法对给定频率 $\omega$ 的谐振信号作估计所得的等式一致,其中谐振信号可以是如下的信号模型

$$h(t) = A\cos\omega t - B\sin\omega t \tag{13.8.6}$$

此事实说明了为什么这种方法给出比 FFT 方法更好的结果:它对数据的估计建立在"各点"的基础上,而不是建立在"每一时间间隔"上,而后者在非均匀取样时会带来严重的误差。

一个非常普遍的情况是,被测量的数据点 $h_i$ 是周期信号和独立(白的)高斯噪声之和。如果我们试图确定这种周期信号的出现或消失,则我们必须对下面的问题给予一个定量的回答:"在功率谱 $P_N(\omega)$ 中一个峰应明显到怎样程度?"。在这个问题中,虚的假设是假设数据也是独立的高斯分布的随机值。罗姆归一化周期图的一个非常好的性质是,虚假设的可行性可以比较严格地验证,下面我们进行讨论。

"归一化"涉及式(13.8.4)分母中因子 $\sigma^2$。斯卡格尔证明在这种归一化下,在一些特定频率和虚假设的前提下,$P_N(\omega)$ 具有均值为 1 的指数分布。换言之,$P_N(\omega)$ 在 $z$ 和 $z+dz$ 之间的概率是 $\exp(-z)dz$。还可以看出,如果我们对某 $M$ 个独立的频率扫描,则不大于 $z$ 值的概率是 $(1-e^{-z})^M$,因此

$$P(>z) \equiv 1-(1-e^{-z})^M \tag{13.8.7}$$

是虚假设的虚警概率,它也是我们所能看到的 $P_N(\omega)$ 中任意尖峰的**显著性水平**。虚警概率的一个小值指出了一个非常明显的周期信号的存在。

为了评估这种显著性,我们需要知道 $M$。事实上,我们观察的频率越多,谱中的尖峰越不明显。(如果观察得足够长,可能什么也没有!)一个典型的过程是,在比较大的频率范围内把 $P_N(\omega)$ 绘画成划分间隔很细

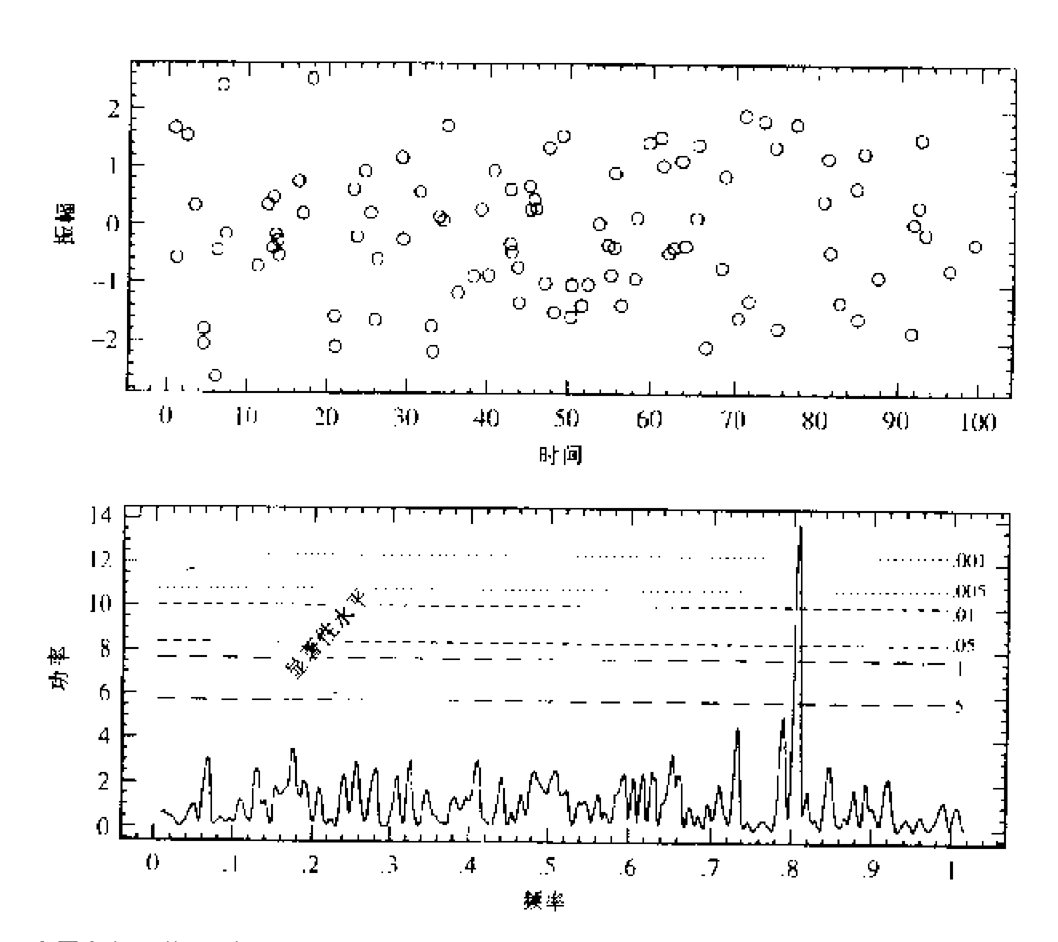
的空间频率的函数。这些频率中有多少个是独立的呢？

在回答这个问题之前，让我们首先看一看 $M$ 需怎样的精确度。我们感兴趣的区间是显著性为一个小数，即远离 1 的地方。所以等式(13.8.7)可以以通过级数展开为

$$P(>z) \approx Me^{-z} \tag{13.8.8}$$

我们看到显著性与 $M$ 成线性比例关系。实际的显著性水平是 0.05 、0.01 和 0.001 等数。在评估显著性中即使存在 ±50% 的误差也是可容忍的，因为所引用的显著性水平通常是根据因子 5 或 10 来划分的，因此我们对 $M$ 的估计不必非常精确。

霍恩(Horne)和柏里那斯(Baliunas)[5]从大量的蒙特卡罗(Monte Carlo)实验中给出了在不同情况下如何确定 $M$ 的取值。通常，$M$ 依赖于取样的频率数，数据点 $N$ 的数值以及它们的细分间隔。当数据点接近等间距划分时，并且当取样的频率"充满"了从零到奈奎斯特频率 $f_c$(式(13.8.2))间的所有频率范围时，可以发现 $M$ 和 $N$ 非常接近。还有，对于随机间隔的数据点或等距间隔的数据点，$M$ 的值没有很大的区别，当用大于奈奎斯特频率的频率范围取样时，$M$ 成比例地增加。只有在这种情况下 $M$ 的取值将明显地不同于等距间隔的情况，即数据点被紧密云集在一起，例如数据聚集成 3 组，那么(我们可以想像)独立的频率数将减少大约 1/3 。




上图中在 0 到 100 的时间范围内 100 个数据是随机分布的。数据中的正弦分量在显著性水平优于 0.001 处被算法鉴别出来。如果 100 个数据点以单位时间间隔均匀地分隔，那么奈奎斯特临界频率将是 0.5 。注意，对非均匀时间间隔分隔的点，在奈奎斯特频率范围内也没有明显的混迭。

图13.8.1 实施罗姆(Lomb)算法的事例

下面的程序基于上述的粗糙但尚能用的规则来计算有效的 $M$ 值，并且假设没有任何重要的数据聚集。在大多数情况下这种计算是足够的。但在一些特殊情况下，如果确定麻烦的话，用简单的蒙特卡罗方法

计算更好的 M 值是不太困难的：用固定数目的数据点以及对应的位置 $t_i$，产生具有高斯（正态）方差分布的合成数据组，对这样的数据组找出 $P_N(\omega)$ 的最大值（用附带的程序），再根据等式（13.8.7）对所得的数据分布拟合求得 $M$。

上页图13.8.1显示了应用上面讨论的方法所得的结果。在上图中，数据点是按时间轴绘出的。数据点的个数是 $N=100$，其随时间 $t$ 的分布是泊松分布。当然，我们看不出任何一点关于含有正弦信号的证据。而下图是按频率 $f=\omega/2\pi$ 绘制的 $P_N(\omega)$。如果数据点是按均匀间隔取样的话，那么获得的奈奎斯特临界频率将是 $f=f_c=0.5$。因为现在我们要搜索的频率范围是临界频率的两倍，又因为对频率 $f$ 过取样的点其 $P_N(\omega)$ 的连续值应是光滑地变化，所以取 $M=2N$。图中水平的虚线和点线从下至上依次表示显著性水平为 0.5、0.1、0.05、0.01、0.005 和 0.001。可以看出，在频率 0.81 处有一个非常显著的尖峰，这正是出现在数据中的正弦波频率。

注意图中附近另外还有两个峰，但都没有超过50%的显著性水平；它也可能是期望被选取的。还有一个值得注意的事实是：那个明显的峰是在**奈奎斯特频率**以上找到的（是正确的），而且在奈奎斯特频率之下不引起任何明显的混迭。这一点对均匀间隔取样的数据是不可能的。在这里成为可能是因为一些数据点的随机时间取样间隔比"平均"取样率对应的时间间隔更紧密，而这就排除了混迭的模糊性。

用程序实施归一化的周期图是很直接的，但是有几点必须记住。我们正在进行的是一个很慢的算法。一般说来，对于 $N$ 个数据点，我们可能希望实验 $2N$ 到 $4N$ 个频率点。在等式（13.8.4）和（13.8.5）中，每组频率和数据点的组合不仅仅是一些加法和乘法，而且有四次调用三角函数；运算很容易就达到 $N^2$ 的几百倍。用递归代替这些三角函数的调用就非常必要，这将使运算速度提高 4 倍。但是这种做法仅在被检测的频率序列是线性序列时才可能。因为这种序列是大多数用户所需要的，所以我们将把这种算法付诸实施。

在这一节的最后我们将叙述一种方法，它近似于估计等式（13.8.4）和（13.8.5），但是具有一定的近似精确度，这种方法比较快，运算量达 $N\log N$[8]，这种快速方法适用于长数据组。

要检测的、独立的最低频率 $f$ 是输入信号时间长度 $\max_i(t_i)-\min_i(t_i)\equiv T$ 的倒数，这个频率能使数据包括在一个完整周期内。再减去数据的平均值，等式（13.8.4）已假设对数据的零频部分不感兴趣——零频对应的恰好是均值。在 FFT 方法中，较高的独立频率将是$1/T$ 的整数倍。因为我们感兴趣的是尖峰出现的统计显著性，因此我们的取样间隔最好比$1/T$ 更精细一些，以便使取样点更接近任何峰的顶点。附带的程序中包含有一个过取样参数，称为 ofac，实际应用中通常取 ofac≥4。我们也必须标明最高频率 $f_{hi}$。$f_{hi}$ 的选择是将它和奈奎斯特频率 $f_c$ 相比，如果在同样的时间间隔 $T$ 内均匀选取 $N$ 个数据点，那么 $f_c=N/2T$。附带的程序中还包括一个输入参数 hifac，它定义为 $f_{hi}/f_c$。由程序返回的不同频率数目 $N_p$ 由下式给出

$$N_p=\frac{\text{ofac}\times\text{hifac}}{2}N \tag{13.8.9}$$

（记住输出数组的长度必须在此范围内。）

程序对三角函数递归采用双精度，并且利用一些三角函数恒等关系以减小舍入误差。如果读者对此感兴趣的话可以仔细推敲。最后要注意的一个细节是，如果 $\bar{x}$ 太大了的话，则舍入误差等式（13.8.7）将无效；但（13.8.8）式在这方法中仍是有效的。

```
#include <math.h>
#include "nrutil.h"
#define TWOPID 6.2831853071795865

void period(float x[], float y[], int n, float ofac, float hifac, float px[],
    float py[], int np, int *nout, int *jmax, float *prob)
```

给定 $n$ 个数据点，横坐标为 x[1..n]（这并不需要均匀间隔），纵坐标为 y[1..n]，并且已知所需的过取样因子 ofac（典型值为4或更大）。程序返回数组 px[1..np]，其频率（不是角频率）顺序增加到 hifac 乘以"平均"的奈奎斯特频率，以及数组 py[1..np]，其值为上面的各频率处的罗姆归一化周期图。数组 x 和 y 维持不变。np 是 px 和 py 的长度，必须足够大才能包含输出结果，否则将出错。程序还返回 jmax，以便 py[jmax]成为数组 py 中的最大元素；程序返回的 prob，是对于随机噪声假设的最大显著性估算值。较小的 prob 表明有一个明显的周期信号出现。

```
{
    void avevar(float data[], unsigned long n, float *ave, float *var);
    int i,j;
    float ave,c,cc,cwtau,effm,expy,pnow,pymax,s,ss,sumc,sumcy,sums,sumsh,
        sumsy,swtau,var,wtau,xave,xdif,xmax,xmin,yy;
    double arg,wtemp,*wi,*wpi,*wpr,*wr;

    wi=dvector(1,n);
    wpi=dvector(1,n);
    wpr=dvector(1,n);
    wr=dvector(1,n);
    *nout=0.5*ofac*hifac*n;
    if (*nout > np) nrerror("output arrays too short in period");
    avevar(y,n,&ave,&var);                   求输入数据的均值和方差
    xmax=xmin=x[1];                          扫描数据，确定横坐标的范围
    for (j=1;j<=n;j++) {
        if (x[j] > xmax) xmax=x[j];
        if (x[j] < xmin) xmin=x[j];
    }
    xdif=xmax-xmin;
    xave=0.5*(xmax+xmin);
    pymax=0.0;
    pnow=1.0/(xdif*ofac);                    起始频率
    for (j=1;j<=n;j++) {                     在每个数据点对三角函数递归进行初始赋值，
        arg=TWOPID*((x[j]-xave)*pnow);           递归以双精度数据进行
        wpr[j] = -2.0*SQR(sin(0.5*arg));
        wpi[j]=sin(arg);
        wr[j]=cos(arg);
        wi[j]=wpi[j];
    }
    for (i=1;i<=(*nout);i++) {               对要估计的频率的主循环
        px[i]=pnow;
        sumsh=sumc=0.0;                      首先，对数据循环得到 τ 和相关的量
        for (j=1;j<=n;j++) {
            c=wr[j];
            s=wi[j];
            sumsh += s*c;
            sumc += (c-s)*(c+s);
        }
        wtau=0.5*atan2(2.0*sumsh,sumc);
        swtau=sin(wtau);
        cwtau=cos(wtau);
        sums=sumc=sumsy=sumcy=0.0;           接着，再对数据循环得到周期图的值
        for (j=1;j<=n;j++) {
            s=wi[j];
            c=wr[j];
            ss=s*cwtau-c*swtau;
            cc=c*cwtau+s*swtau;
            sums += ss*ss;
            sumc += cc*cc;
            yy=y[j]-ave;
            sumsy += yy*ss;
            sumcy += yy*cc;
            wr[j]=((wtemp=wr[j])*wpr[j]-wi[j]*wpi[j])+wr[j];    对三角函数递归进行
            wi[j]=(wi[j]*wpr[j]+wtemp*wpi[j])+wi[j];            修正
        }
        py[i]=0.5*(sumcy*sumcy/sumc+sumsy*sumsy/sums)/var;
        if (py[i] >= pymax) pymax=py[(*jmax=i)];
            pnow += 1.0/(ofac*xdif);          下一个频率
        }
        expy=exp(-pymax);                     求最大值的统计显著性
        effm=2.0*(*nout)/ofac;
        *prob=effm*expy;
        if (*prob > 0.01) *prob=1.0-pow(1.0-expy,effm);
        free_dvector(wr,1,n);
        free_dvector(wpr,1,n);
```

```
    free_dvector(wpi,1,n);
    free_dvector(wi,1,n);
}
```

### 13.8.1 罗姆周期图快速计算

下面我们说明怎样快速近似计算等式(13.8.4)和(13.8.5),但是这种近似却能达到想要的精度,并且其运算量为 $N_p\log N_p$ 的数量级。此方法要用到 FFT,但决不是数据的 FFT 周期图,它实际上对等式(13.8.4)和(13.8.5),即罗姆归一化周期图的估计,它完全具有 FFT 周期图的优点和弱点。由于普瑞斯(Press)和赖比克(Rybicki)[6]的贡献,使得将罗姆方法运用到$10^6$个数据点的计算成为可能,它比用式(13.8.4)和(13.8.5)式对数据点为 60 到 100 时的直接计算还要快。

注意,等式(13.8.5)和(13.8.4)中的三角函数求和可以化为四个简单的求和。如果我们定义

$$S_h \equiv \sum_{j=1}^{N}(h_j - \bar{h})\sin(\omega t_j) \qquad C_h \equiv \sum_{j=1}^{N}(h_j - \bar{h})\cos(\omega t_j) \tag{13.8.10}$$

和

$$S_2 \equiv \sum_{j=1}^{N}\sin(2\omega t_j) \qquad C_2 \equiv \sum_{j=1}^{N}\cos(2\omega t_j) \tag{13.8.11}$$

那么

$$\begin{aligned}
&\sum_{j=1}^{N}(h_j - \bar{h})\cos\omega(t_j - \tau) = C_h\cos\omega\tau + S_h\sin\omega\tau \\
&\sum_{j=1}^{N}(h_j - \bar{h})\sin\omega(t_j - \tau) = S_h\cos\omega\tau - C_h\sin\omega\tau \\
&\sum_{j=1}^{N}\cos^2\omega(t_j - \tau) = \frac{N}{2} + \frac{1}{2}C_2\cos(2\omega\tau) + \frac{1}{2}S_2\sin(2\omega\tau) \\
&\sum_{j=1}^{N}\sin^2\omega(t_j - \tau) = \frac{N}{2} - \frac{1}{2}C_2\cos(2\omega\tau) - \frac{1}{2}S_2\sin(2\omega\tau)
\end{aligned} \tag{13.8.12}$$

现在注意,如果时间 $t_j$ 是均匀间隔的,那么 $S_h$、$C_h$、$S_2$和 $C_2$可用两个复数的 FFT 计算,结果代入式(13.8.12),再由式(13.8.12)可估算等式(13.8.5)和(13.8.4)。因此余下的问题是,在非均匀间隔的数据点处对等式(13.8.10)和(13.8.11)进行估算。

答案是采用内插或倒插值,我们称倒插值为**扩展插值**。插值,如同经典描述的那样,使用在一规则网格上的几个函数值来构造任意点处的精确近似值。扩展插值恰好相反,它是用规则网络上的几个函数值代替任意点上的函数值,并使网格上各点的和是原来任意点之和的精确近似。

不难看出,扩展插值的权函数等价于一般插值的权函数。假设要插值的函数 $h(t)$仅在离散点(非均匀间隔)上知道其值$h(t_i)\equiv h_i$,而函数 $g(t)$(例如它可以是$\cos\omega t$)在任意处都能估计其值。令 $\hat{t}_k$ 为一个规则网格上的均匀间隔的点序列。那么拉格朗日插值(第3.1节)给出下面形式的一个近似:

$$g(t) \approx \sum_k w_k(t)g(\hat{t}_k) \tag{13.8.13}$$

其中,$w_k(t)$是插值的权重。现在让我们通过下式来估算我们感兴趣的和式:

$$\sum_{j=1}^{N} h_j\, g(t_j) \approx \sum_{j=1}^{N} h_j\Big[\sum_k w_k(t_j)g(\hat{t}_k)\Big] = \sum_k\Big[\sum_{j=1}^{N} h_j w_k(t_j)\Big]g(\hat{t}_k) \equiv \sum_k \hat{h}_k\, g(\hat{t}_k) \tag{13.8.14}$$

这里,$\hat{h}_k \equiv \sum_j h_j\, w_k(t_j)$。注意等式(13.8.14)用规则网格上的和值代替原先的和值。还请注意等式(13.8.13)的精确度取决于函数 $g$ 划分的网格精细程度,而和时间点 $t_j$ 的间隔及函数 $h$ 无关。等式(13.8.14)的精确度也有这种性质。

因此这种快速估计方法的一般轮廓是:(i)选择一个足够大的网格能容纳想要的过取样因子,还要能在感兴趣的最高频率的每半个波长处有几个外插点;(ii)在网格上外插值 $h_i$,并且进行 FFT,这就给出等式

(13.8.10)中的 $S_h$ 和 $C_h$；(iii)将常数值1外插到另一个网格上，并进行 FFT；接着通过一些运算计算出等式(13.8.11)中的 $S_2$和 $C_2$；(iv)依次估算(13.8.12)、(13.8.5)和(13.8.4)三个等式。

为了更有效地实施这种算法，还有一些其它的技巧。从程序中读者可以推测出大多数的技巧，但我们将说明以下几点：(a)用频率$2\omega$处的变换值代替$\omega$处的值的技巧是，将时间域的数据扩展为2倍，接着双倍覆盖初始的长度。(这种技巧归于托基(Tukey))。在程序中，这将以模函数的形式出现。(b)从等式(13.8.5)的左边到各种所需的 $\omega\tau$ 的三角函数，都用到了三角函数的恒等式。C 语言中标识符如 cwt、hs2wt 代表了量 $\cos\omega\tau$ 和 $\frac{1}{2}\sin(2\omega\tau)$。(c)函数 **spread** 在围绕任意点的 $M$ 个靠近中心网格点处实施扩展插值，它的附加程序用非常有效的方法估算了拉格朗日插值多项式的系数。

```
#include <math.h>
#include "nrutil.h"
#define MOD(a,b)   while(a >= b) a -= b;          只是正数
#define MACC 4                                     最高频率每1/4圈的插值点数

void fasper(float x[], float y[], unsigned long n, float ofac, float hifac,
    float wk1[], float wk2[], unsigned long nwk, unsigned long *nout,
    unsigned long *jmax, float *prob)
```

给定 n 个数据点的横坐标 x[1..n](并不需要是均匀间隔)和纵坐标 y[1..n]，还给定期望的过取样因子 ofac(典型值为4或更大)此程序返回数组 wk1[1..nwk]，它是 nout 个递增的频率(不是角频率)序列，最大值为 hifac 乘以"均匀"的奈奎斯特频率；还返回数组 wk2[1..nwk]，其值是在这些频率处罗姆归一化周期图的值。数组 x 和 y 维持不变，数组 wk1和 wk2的维数 nwk 必须足够大，以便有中介计算空间，否则会出错。程序还返回 jmax，以使 wk2[jmax]为数组 wk2中的最大元素，并且还返回 prob，它是在随机噪声假设下最大显著性的估算值。较小的 prob 值说明有一个明显的周期信号出现。

```
{
    void avevar(float data[], unsigned long n, float *ave, float *var);
    void realft(float data[], unsigned long n, int isign);
    void spread(float y, float yy[], unsigned long n, float x, int m);
    unsigned long j,k,ndim,nfreq,nfreqt;
    float ave,ck,ckk,cterm,cwt,den,df,effm,expy,fac,fndim,hc2wt;
    float hs2wt,hypo,pmax,sterm,swt,var,xdif,xmax,xmin;
    *nout=0.5*ofac*hifac*n;
    nfreqt=ofac*hifac*n*MACC;                   将FFT的大小定为大于
    nfreq=64;                                        nfreqt 的最小幂次
    while (nfreq < nfreqt) nfreq <<= 1;
    ndim=nfreq << 1;
    if (ndim > nwk) nrerror("workspaces too small in fasper");
    avevar(y,n,&ave,&var);                      计算数据的均值、方差和范围
    xmin=x[1];
    xmax=xmin;
    for (j=2;j<=n;j++) {
        if (x[j] < xmin) xmin=x[j];
        if (x[j] > xmax) xmax=x[j];
    }
    xdif=xmax-xmin;
    for (j=1;j<=ndim;j++) wk1[j]=wk2[j]=0.0;   置工作空间为0
    fac=ndim/(xdif*ofac);
    fndim=ndim;
    for (j=1;j<=n;j++) {                         外插数据于工作空间
        ck=(x[j]-xmin)*fac;
        MOD(ck,fndim)
        ckk=2.0*(ck++);
        MOD(ckk,fndim)
        ++ckk;
        spread(y[j]-ave,wk1,ndim,ck,MACC);
        spread(1.0,wk2,ndim,ckk,MACC);
    }
    realft(wk1,ndim,1);                          进行快速傅立叶变换
    realft(wk2,ndim,1);
```

```
    df=1.0/(xdif*ofac);
    pmax = -1.0;
    for (k=3,j=1;j<=(*nout);j++,k+=2) {        对每个频率计算罗姆值
        hypo=sqrt(wk2[k]*wk2[k]+wk2[k+1]*wk2[k+1]);
        hc2wt=0.5*wk2[k]/hypo;
        hs2wt=0.5*wk2[k+1]/hypo;
        cwt=sqrt(0.5+hc2wt);
        swt=SIGN(sqrt(0.5-hc2wt),hs2wt);
        den=0.5*n+hc2wt*wk2[k]+hs2wt*wk2[k+1];
        cterm=SQR(cwt*wk1[k]+swt*wk1[k+1])/den;
        sterm=SQR(cwt*wk1[k+1]-swt*wk1[k])/(n-den);
        wk1[j]=j*df;
        wk2[j]=(cterm+sterm)/(2.0*var);
        if (wk2[j] > pmax) pmax=wk2[(*jmax=j)];
    }
    expy=exp(-pmax);                          估计最大峰值的显著性
    effm=2.0*(*nout)/ofac;
    *prob=effm*expy;
    if (*prob > 0.01) *prob=1.0-pow(1.0-expy,effm);
}
```

```
#include "nrutil.h"

void spread(float y, float yy[], unsigned long n, float x, int m)
    给定数值 yy[1..n]扩展插值 y 到 m 个实际的数组元素中,这些元素非常接近"虚构"(也就是可能不是整数)的数组
    元素 x。所用的权重是拉格朗日插值多项式的系数。
{
    int ihi,ilo,ix,j,nden;
    static int nfac[11]={0,1,1,2,6,24,120,720,5040,40320,362880};
    float fac;

    if (m > 10) nrerror("factorial table too small in spread");
    ix=(int)x;
    if (x == (float)ix) yy[ix] += y;
    else {
        ilo=LMIN(LMAX((long)(x-0.5*m+1.0),1),n-m+1);
        ihi=ilo+m-1;
        nden=nfac[m];
        fac=x-ilo;
        for (j=ilo+1;j<=ihi;j++) fac *= (x-j);
        yy[ihi] += y*fac/(nden*(x-ihi));
        for (j=ihi-1;j>=ilo;j--) {
            nden=(nden/(j+1-ilo))*(j-ihi);
            yy[j] += y*fac/(nden*(x-j));
        }
    }
}
```

## 参考文献和进一步读物:


Lomb, N. R. 1976, *Astrophysics and Space Science*, vol. 39, pp. 447~462. [1]

Barning F, J. M. 1963, *Bulletin of the Astronomical Institutes of the Netherlands*, vol. 17, pp. 22~28. [2]

Vahcek, P. 1971, *Astrophysics and Space Science*, vol. 12, pp. 10~33. [3]

Scargle, J. D. 1982, *Astrophysical Journal*, vol. 263, pp. 835~853. [4]

Home, J. H., and Baliunas, S. L. 1986, *Astrophysical Journal*, vol. 302, pp. 757~763. [5]

Press, W. H., and Rybicki, G. B. 1989, *Astrophysical Journal*, vol. 338, pp. 277~280. [6]

## 13.9 用 FFT 计算傅里叶积分

在许多情况下，人们想计算下面积分式的精确数值

$$I = \int_a^b e^{i\omega t} h(t) dt, \tag{13.9.1}$$

或等价的实部和虚部

$$I_c = \int_a^b \cos(\omega t) h(t) dt \quad I_s = \int_a^b \sin(\omega t) h(t) dt \tag{13.9.2}$$

并且人们要对许多不同的 $\omega$ 值的估算此积分式。在很多感兴趣的情况下，$h(t)$ 通常是一个光滑的函数，但并不需以 $[a,b]$ 为周期，也并不要求它在 $a$ 和 $b$ 点处之值为 0。直观上很明显，由于 FFT 的巨大"魔力"，它应该能用于此问题。但是我们将看到，这样做是件令人感到吃惊的非常微妙的事情。

让我们首先用最直观的方法来考虑这个问题，并且看看困难在哪里。将间隔 $[a,b]$ 分成 $M$ 个小的间隔，其中 $M$ 是一个大的整数，并且定义

$$\Delta \equiv \frac{b-a}{M}, \quad t_j \equiv a + j\Delta, \quad h_j \equiv h(t_j), \quad j = 0, \ldots, M \tag{13.9.3}$$

注意 $h_0 \equiv h(a)$，$h_M = H(b)$，有 $M+1$ 个 $h_j$ 值，我们可以用下面的和式近似积分 $I$

$$I \approx \Delta \sum_{j=0}^{M-1} h_j \exp(i\omega t_j) \tag{13.9.4}$$

它具有一阶的精确性(如果我们将 $h_j$ 和 $t_j$ 置于划分间隔的中央，我们便能到二阶的精度)。现在对于 $\omega$ 和 $M$ 的一些值，等式(13.9.4)中的和式可以进行离散傅里叶变换，即 DFT，并且能通过快速傅里叶(FFT)算法进行计算。特别地，我们可以选 $M$ 为 2 的幂次，并且通过下式定义一组特殊的 $\omega$ 值：

$$\omega_m \Delta \equiv \frac{2\pi m}{M} \tag{13.9.5}$$

其中 $m$ 的取值为 $m=0,1,\ldots,M/2-1$，那么等式(13.9.4)便成为：

$$I(\omega_m) \approx \Delta e^{i\omega_m a} \sum_{j=0}^{M-1} h_j e^{2\pi i m j/M} = \Delta e^{i\omega_m a} [\mathrm{DFT}(h_0 \ldots h_{M-1})]_m \tag{13.9.6}$$

虽然等式(13.9.6)简单清楚，但我们郑重地强调不推荐使用此式，因为它有可能导致错误的结果。

问题在于积分式(13.9.1)的振荡特性，如果 $h(t)$ 确实光滑，并且 $\omega$ 足够大，以致在间隔 $[a,b]$ 中有几个周期——实际上等式(13.9.5)中的 $\omega_m$ 恰好给出 $m$ 个周期——那么积分值 $I$ 通常很小，如此小以致很容易被一阶甚至二阶的截断误差所淹没。还有，如果被积函数不是振荡的，出现在误差项中的特征"小参数"将不是 $\Delta/(b-a)=1/M$，而是 $\omega\Delta$。对于在 DFT(参看等式13.9.5)的奈奎斯特间隔中的 $\omega$ 值，此值可以大到 $\pi$。其结果是当 $\omega$ 增加时，等式(13.9.6)出现系统误差。

一个有益的练习就是用一个可以解析地求积函数对等式(13.9.6)验证，就可以看出它的精确程度如何。我们建议读者试一试。

现在，让我们看一个更复杂的情况。给出采样点 $h_j$，我们可以利用 $h_j$ 值附近的插值法，以使在间隔 $[a,b]$ 内各处，逼近函数 $h(t)$。最简单的情况就是线性插值，有两个最接近的 $h_j$ 值，一个在左边，一个在右边。高阶的插值，例如三阶插值，左边有两点，右边有两点——除了数据中第一个间隔和最后间隔的情况，此时，插值需在一边用三个 $h_j$ 值，另一边则用一个。

这种插值方式的公式是独立变量 $t$ 的(分段)多项式，但是系数却是 $h_j$ 的线性函数。尽管通常人们不从这个角度考虑，插值还是可以看成一系列核函数(只依赖于插值方式)和样本值(仅依赖于函数值)乘积之和的近似函数。我们可以写成下式

$$h(t) \approx \sum_{j=0}^{M} h_j \psi\left(\frac{t-t_j}{\Delta}\right) + \sum_{j=\text{端点}} h_j \varphi_j\left(\frac{t-t_j}{\Delta}\right) \tag{13.9.7}$$

这里，$\psi(s)$是内点的核函数：如果 $s$ 足够负或足够正，那么它是零；仅当 $s$ 处于和 $h_j$ 的乘积确实是用来插值时，$\psi(s)$才不为零。因为正好在样本点上的的插值应该给出采样函数值，所以我们永远有 $\psi(0)=1, \psi(m)=0, m=\pm1,\pm2,\ldots$ 对于线性插值，$\psi(s)$是分段线性的，当 s 在$(-1,0)$时，$\psi(s)$从 0 升到 1；当 $s$ 在$(0,1)$时，$\psi(s)$又回到 0 。更高阶的插值 $\psi(s)$由拉格朗日插值多项式的几段构成。在整数 $s$ 处，即各段连接处，由于插值点是离散地变化的，$\psi(s)$有不连续的导数。

正如我们已经看到的那样，最靠近 $a$ 和 $b$ 的小间隔要求不同的(非中心的)插值公式。这可以从等式(13.9.7)中第二项得到反映，其中有一个特殊的端点核函数 $\varphi_j(s)$。实际上，由于一些下面我们将看得更明白的原因，我们将所有的点包括在第一个和式里(核函数为 $\psi$)，而 $\varphi_j$实际上是真正的端点核函数和内核函数 $\psi$ 的差值。将任何阶插值的 $\varphi_j$写下来是一个枯燥的但很直观的工作，每个 $\varphi_j$由连接在一起的各段拉格朗日插值多项式的差值构成。

现在我们将积分运算 $\int_a^b dt\ \mathrm{esp}(i\omega t)$ 运用到等式(13.9.7)两边，交换积分和求和的顺序，并且对第一项求和进行积分变换 $s=(t-t_j)/\Delta$，对第二项求和进行积分变换 $s=(t-a)/\Delta$，结果是

$$I \approx \Delta e^{i\omega a}\left[W(\theta)\sum_{j=0}^{M} h_j e^{ij\theta} + \sum_{j=\text{端点}} h_j \alpha_j(\theta)\right] \qquad (13.9.8)$$

这里 $\theta \equiv \omega\Delta$，函数 $W(\theta)$和 $\alpha_j(\theta)$由下式定义：

$$W(\theta) \equiv \int_{-\infty}^{\infty} ds\, e^{i\theta s}\psi(s) \qquad (13.9.9)$$

$$\alpha_j(\theta) \equiv \int_{-\infty}^{\infty} ds\, e^{i\theta s}\varphi_j(s \quad j) \qquad (13.9.10)$$

非常重要的一点是，对任何插值方式等式(13.9.9)和等式(13.9.10)可以解析地估算。等式(13.9.8)表示了一个将“端点校正”运用到求和式的算法，此求和式可以用 FFT 计算，由此给出高阶精度的结果。

我们将只考虑在左右对称的插值，该对称意味着

$$\varphi_{M-j}(s) = \varphi_j(-s) \qquad \alpha_{M-j}(\theta) = e^{i\theta M}\alpha_j^*(\theta) = e^{i\omega(b-a)}\alpha_j^*(\theta) \qquad (13.9.11)$$

其中的星号代表了复共轭。还有 $\psi(s)=\psi(-s)$意味 $W(\theta)$是实数。

现在让我们看等式(13.9.8)中的第一个和式，我们将用 FFT 方法计算。为了进行 FFT，首先选择 $N$，使得 $N\geqslant M+1$，并且 $N$ 为 2 的整数幂次。(注意 $M$ 不必是 2 的幂次，因此 $M=N-1$ 是容许的)。如果 $N>M+1$，定义 $h_j\equiv0$，$M+1<j\leqslant N-1$，即对数组 $h_j$ 进行“零元填充”以使 $j$ 的取值范围是 $0\leqslant j\leqslant N-1$。那么在下式确定的特殊值 $\omega=\omega_n$ 下，可对和式进行 DFT。

$$\omega_n\Delta \equiv \frac{2\pi n}{N} \equiv \theta \qquad n=0,1,\ldots,\frac{N}{2}-1 \qquad (13.9.12)$$

对于固定的 $M$，$N$ 选择得越大，频域的采样越细，另一方面，$M$ 值又确定了最高的取样频率，因为当 $M$ 增加时，$\Delta$ 减小(等式(13.9.3))，而最大的 $\omega\Delta$ 值总是在 $\pi$ 以下(等式(13.9.12))。一般说来，过取样因子至少取为 4 比较有利，即 $N>4M$(参看下面)，最后我们可将等式(13.9.8)写成如下的形式

$$\begin{aligned} I(\omega_n) = \Delta e^{i\omega_n a}\{ & W(\theta)[\mathrm{DFT}(h_0\ldots h_{N-1})]_n \\ & + \alpha_0(\theta)h_0 + \alpha_1(\theta)h_1 + \alpha_2(\theta)h_2 + \alpha_3(\theta)h_3 + \ldots \\ & + e^{i\omega(b-a)}[\alpha_0^*(\theta)h_M + \alpha_1^*(\theta)h_{M-1} + \alpha_2^*(\theta)h_{M-2} + \alpha_3^*(\theta)h_{M-3} + \ldots]\} \end{aligned} \qquad (13.9.13)$$

对于三阶(或更低)的插值多项式，上式中明显地写出来的各项大多是非零值，而用省略号略去的各项可以被忽略，因此我们只需要由等式(13.9.9)和(13.9.10)计算求得函数 $W$，$\alpha_0$，$\alpha_1$、$\alpha_2$、$\alpha_3$的明显表达式。对梯形(二阶)和立方(四阶)的情况我们已经计算出了表达式，下面是它们的结果，并且给出了对于小 $\theta$ 时，它们的幂级数展开的前几项：

梯形阶数：

$$W(\theta) = \frac{2(1-\cos\theta)}{\theta^2} \approx 1 - \frac{1}{12}\theta^2 + \frac{1}{360}\theta^4 - \frac{1}{20160}\theta^6$$

$$\alpha_0(\theta) = -\frac{(1-\cos\theta)}{\theta^2} + i\frac{(\theta-\sin\theta)}{\theta^2}$$
$$\approx -\frac{1}{2} + \frac{1}{24}\theta^2 - \frac{1}{720}\theta^4 - \frac{1}{40320}\theta^6 - i\theta\left(\frac{1}{6} - \frac{1}{120}\theta^2 + \frac{1}{5040}\theta^4 - \frac{1}{362880}\theta^6\right)$$
$$\alpha_1 = \alpha_2 = \alpha_3 = 0$$

立方阶数：

$$W(\theta) = \left(\frac{6+\theta^2}{3\theta^4}\right)(3 - 4\cos\theta - \cos 2\theta) \approx 1 - \frac{11}{720}\theta^4 + \frac{23}{15120}\theta^6$$
$$\alpha_0(\theta) = \frac{(-42-5\theta^2) + (6+\theta^2)(8\cos\theta - \cos 2\theta)}{6\theta^4} + i\frac{(-12\theta + 6\theta^3) + (6+\theta^2)\sin 2\theta}{6\theta^4}$$
$$\approx -\frac{2}{3} + \frac{1}{45}\theta^2 + \frac{103}{15120}\theta^4 - \frac{169}{226800}\theta^6 + i\theta\left(\frac{2}{45} + \frac{2}{105}\theta^2 - \frac{8}{2835}\theta^4 + \frac{86}{467775}\theta^6\right)$$
$$\alpha_1(\theta) = \frac{14(3-\theta^2) - 7(6+\theta^2)\cos\theta}{6\theta^4} + i\frac{30\theta - 5(6+\theta^2)\sin\theta}{6\theta^4}$$
$$\approx \frac{7}{24} - \frac{7}{180}\theta^2 + \frac{5}{3456}\theta^4 - \frac{7}{259200}\theta^6 + i\theta\left(\frac{7}{72} - \frac{1}{168}\theta^2 + \frac{11}{72576}\theta^4 - \frac{13}{5987520}\theta^6\right)$$
$$\alpha_2(\theta) = \frac{-4(3-\theta^2) + 2(6+\theta^2)\cos\theta}{3\theta^4} + i\frac{-12\theta + 2(6+\theta^2)\sin\theta}{3\theta^4}$$
$$\approx -\frac{1}{6} + \frac{1}{45}\theta^2 - \frac{5}{6048}\theta^4 + \frac{1}{64800}\theta^6 - i\theta\left(-\frac{7}{90} + \frac{1}{210}\theta^2 - \frac{11}{90720}\theta^4 + \frac{13}{7484400}\theta^6\right)$$
$$\alpha_3(\theta) = \frac{2(3-\theta^2) - (6+\theta^2)\cos\theta}{6\theta^4} + i\frac{6\theta - (6+\theta^2)\sin\theta}{6\theta^4}$$
$$\approx \frac{1}{24} - \frac{1}{180}\theta^2 + \frac{5}{24192}\theta^4 - \frac{1}{259200}\theta^6 + i\theta\left(\frac{7}{360} - \frac{1}{840}\theta^2 + \frac{11}{362880}\theta^4 - \frac{13}{29937600}\theta^6\right)$$

下面的程序 **dftcor**，对立方情况实施端点校正。给定输入值 $\omega$、$\Delta$、$a$、$b$ 和具有八个值的 $h_0,\ldots,h_3$，$h_{M-3},\ldots,h_M$ 的数组，它返回等式(13.9.13)中的端点校正的实部和虚部以及因子 $W(\theta)$。程序的代码比较臃肿，这完全是由于上面的表达式很复杂。上面的表达式中还放弃了 $\theta$ 的高次幂项，因此即使校正项只需单精度时，还是有必要用双精度计算等式的右边。对小值 $\theta$ 有必要采用级数展开。$\theta$ 的最佳截取值是由机器的字长决定，但是也可以用实验确定它。在最大值处，这两种方法相对于机器精度给出等价的结果。

```
#include <math.h>

void dftcor(float w, float delta, float a, float b, float endpts[],
    float *corre, float *corim, float *corfac)
```

对于用 DFT 来近似求积分，此程序用来计算乘以 DFT 的校正因子和要加于端点的校正项。输入是角频率 $\omega$，步长 delta，积分的上、下限 a 和 b，而数组 endpts 包含最初和最后4个函数值。校正因子 $W(\theta)$ 以 corfac 返回，端点校正项的实部和虚部分别以 corre 和 corim 返回。

```
{
    void nrerror(char error_text[]);
    float a0i,a0r,a1i,a1r,a2i,a2r,a3i,a3r,arg,c,cl,cr,s,sl,sr,t;
    float t2,t4,t6;
    double cth,ctth,spth2,sth,sth4i,stth,th,th2,th4,tmth2,tth4i;

    th=w*delta;
    if (a >= b || th < 0.0e0 || th > 3.01416e0) nrerror("bad arguments to dftcor");
    if (fabs(th) < 5.0e-2) {        使用级数
        t=th;
        t2=t*t;
        t4=t2*t2;
        t6=t4*t2;
        *corfac=1.0-(11.0/720.0)*t4+(23.0/15120.0)*t6;
        a0r=(-2.0/3.0)+t2/45.0+(103.0/15120.0)*t4-(169.0/226800.0)*t6;
        a1r=(7.0/24.0)-(7.0/180.0)*t2+(5.0/3456.0)*t4-(7.0/259200.0)*t6;
        a2r=(-1.0/6.0)+t2/45.0-(5.0/6048.0)*t4+t6/64800.0;
        a3r=(1.0/24.0)-t2/180.0+(5.0/24192.0)*t4-t6/259200.0;
        a0i=t*(2.0/45.0+(2.0/105.0)*t2-(8.0/2835.0)*t4+(86.0/467775.0)*t6);
        a1i=t*(7.0/72.0-t2/168.0+(11.0/72576.0)*t4-(13.0/5987520.0)*t6);
        a2i=t*(-7.0/90.0+t2/210.0-(11.0/90720.0)*t4+(13.0/7484400.0)*t6);
        a3i=t*(7.0/360.0-t2/840.0+(11.0/362880.0)*t4-(13.0/29937600.0)*t6);
```

```
    } else {                        Use trigonometric formulas in double precision.
        cth=cos(th);
        sth=sin(th);
        ctth=cth*cth-sth*sth;
        stth=2.0e0*sth*cth;
        th2=th*th;
        th4=th2*th2;
        tmth2=3.0e0-th2;
        spth2=6.0e0+th2;
        sth4i=1.0/(6.0e0*th4);
        tth4i=2.0e0*sth4i;
        *corfac=tth4i*spth2*(3.0e0-4.0e0*cth+ctth);
        a0r=sth4i*(-42.0e0+5.0e0*th2+spth2*(8.0e0*cth-ctth));
        a0i=sth4i*(th*(-12.0e0+6.0e0*th2)+spth2*stth);
        a1r=sth4i*(14.0e0*tmth2-7.0e0*spth2*cth);
        a1i=sth4i*(30.0e0*th-5.0e0*spth2*sth);
        a2r=tth4i*(-4.0e0*tmth2+2.0e0*spth2*cth);
        a2i=tth4i*(-12.0e0*th+2.0e0*spth2*sth);
        a3r=sth4i*(2.0e0*tmth2-spth2*cth);
        a3i=sth4i*(6.0e0*th-spth2*sth);
    }
    cl=a0r*endpts[1]+a1r*endpts[2]+a2r*endpts[3]+a3r*endpts[4];
    sl=a0i*endpts[1]+a1i*endpts[2]+a2i*endpts[3]+a3i*endpts[4];
    cr=a0r*endpts[8]+a1r*endpts[7]+a2r*endpts[6]+a3r*endpts[5];
    sr = -a0i*endpts[8]-a1i*endpts[7]-a2i*endpts[6]-a3i*endpts[5];
    arg=w*(b-a);
    c=cos(arg);
    s=sin(arg);
    *corre=cl+c*cr-s*sr;
    *corim=sl+s*cr+c*sr;
}
```

因为 **dfccor** 的用法有时让人困惑，我们也给出一个说明性的程序 **dftint**，它针对一般的 $a$、$b$、$\omega$ 和 $h(t)$。利用 **dftcor** 计算等式(13.9.1)，此程序中有几点值得注意：参数 M 和 NDFT 对应于上面讨论的 $M$ 和 $N$，仅当 $a$ 或 $b$ 或 $h(t)$ 发生变化时，通过连续调用，我们重新计算傅里叶变换。

因为 **dftint** 是用来计算满足 $\omega\Delta<\pi$ 的任何 $\omega$ 值的情况，而不仅仅是由 DFT(等式(13.9.12))返回的特殊值，所以我们在 DFT 谱上进行 MPOL 阶多项式插值，还有一点要提醒的是为了使这种插值精确，必须选一个大的过取样因子($N\gg M$)。在插值后，我们将 **dftcor** 中的端点校正加上，它可以在任何 $\omega$ 值处估算。

尽管 **dftcor** 非常好用，但 **dftint** 仅起说明的作用。它不是一个适于所有目标的程序，因为它不能将其参数 M、NDFT、MPOL 或它的插值方式用到任何特殊函数 $h(t)$ 中，对读者自己的实际情况必须进行实验。

```
#include <math.h>
#include "nrutil.h"
#define M 64                    子间隔的个数
#define NDFT 1024               FFT 的长度(2的幂次)                          (这三个量用户
#define MPOL 6                  插值多项式阶数，用于从 FFT 中获得所要的频率     必须进行优化)
#define TWOPI (2.0 * 3.14159264)

void dftint(float (*func)(float), float a, float b, float w, float *cosint, float *sinint)
    这是一个说明如何使用程序 dftcor 的范例程序。用户应提供一个外函数 func，它返回量 h(t)，程序以 cosint 返回积
    分 ∫_a^b cos(ωt)h(t)dt 值，以 sinint 返回积分 ∫_a^b sin(ωt)h(t)dt 的值。
{
    void dftcor(float w, float delta, float a, float b, float endpts[],
        float *corre, float *corim, float *corfac);
    void polint(float xa[], float ya[], int n, float x, float *y, float *dy);
    void realft(float data[], unsigned long n, int isign);
    static int init=0;
    int j, nn;
```

```
    static float aold = -1.e30,bold= -1.e30,delta,( * funcold)(float);
    static float data[NDFT+1],endpts[9];
    float c,cdft,cerr,corfac,corim,corre,en,s;
    float sdft,serr, * cpol, * spol, * xpol;

    cpol = vector(1,MPOL);
    spol = vector(1,MPOL);
    xpol = vector(1,MPOL);
    if (init != 1 || a != aold || b != bold || func != funcold) {      是否需要初始赋值或只改变ω?
        init=1;
        aold=a;
        bold=b;
        funcold=func;
        delta=(b-a)/M;
        for (j=1;j<=M+1;j++)                                          将函数值装入数据组
            data[j]=( * func)(a+(j-1) * delta);
        for (j=M+2;j<=NDFT;j++)                                       对数据组的其余点进行零元填充
            data[j]=0.0;
        for (j=1;j<=4;j++){                                           对端点赋值
            endpts[j]=data[j];
            endpts[j+4]=data[M-3+j];
        }
        realft(data,NDFT,1);                          realft 在data[2]中返回的是对应于ω_{N/2}无用的值.实际上,
                                                      这个元素包含对应ω_0的虚部,所以其值为0
    :
                                                 对想要的频率在 DFT 结果上插值。如果频率是ω_n,即量 en 是整
                                                 数,则 cdft=data[2 * en-1],sdft=data[2 * en],并且可以省略
                                                 插值
    en=w * delta * NDFT/TWOPI+1.0;
    nn=IMIN(IMAX((int)(en-0.5 * MPOL+1.0),1),NDFT/2-MPOL+1);           插值最左边的点
    for (j=1;j<=MPOL;j++,nn++) {
        cpol[j]=data[2 * nn-1];
        spol[j]=data[2 * nn];
        xpol[j]=nn;
    }
    polint(xpol,cpol,MPOL,en,&cdft,&cerr);
    polint(xpol,spol,MPOL,en,&sdft,&serr);
    dftcor(w,delta,a,b,endpts,&corre,&corim,&corfac);                  求端点校正值和乘积因子W(θ)
    cdft *= corfac;
    sdft *= corfac;
    cdft += corre;
    sdft += corim;

    c=delta * cos(w * a);
    s=delta * sin(w * a);                                              最后乘以Δ和exp(iωa)
    * cosint=c * cdft-s * sdft;
    * sinint=s * cdft+c * sdft;
    free_vector(cpol,1,MPOL);
    free_vector(spol,1,MPOL);
    free_vector(xpol,1,MPOL);
}
```

有时候人们只对等式(13.9.5)中的离散频率$\omega_m$感兴趣,它是在区间$[a,b]$中有整数间隔处的点。对光滑的$h(t)$,在这些$\omega$处的积分$I$其幅度可比在这些$\omega$之间频率处的积分幅度小很多,因为小间隔内两半的积分可能相互抵销。(这也是为了插值的精确性我们必须过取样的原因:$I(\omega)$在$\omega_m$值附近处是小幅度的振荡值)。如果要求在这些$\omega_m$处的积分值,而不采用大数目(可能不精确)的插值,那么必须使$N$为$M$的整数倍数(比较式(13.9.5)和式(13.9.12)),在我们用上面实行的方法中,$N$必须至少是$M+1$,因此最小的这样的乘数是$2M$,导致因子~2的不必要计算。或者人们可以导出类似等式(13.9.13)的公式,但是从 DFT

中略去最后的样本函数 $h_M - h(b)$，而将它包括在 $h_M$ 的端点校正项中。那么就可以设 $M=N$(2 的整数幂)，并且在没有附加开销下得到(13.9.5)式的特定频率。其修正公式如下

$$\begin{aligned} I(\omega_m) = \Delta e^{i\omega_m a}\{ & W(\theta)[\mathrm{DFT}(h_0 \ldots h_{M-1})]_m \\ & + \alpha_0(\theta)h_0 + \alpha_1(\theta)h_1 + \alpha_2(\theta)h_2 + \alpha_3(\theta)h_3 \\ & + e^{i\omega(b-a)}[A(\theta)h_M + \alpha_1^*(\theta)h_{M-1} + \alpha_2^*(\theta)h_{M-2} - \alpha_3^*(\theta)h_{M-3}]\} \end{aligned} \tag{13.9.14}$$

其中 $\theta \equiv \omega_m \Delta$，对于梯形情况 $A(\theta)$ 由下式给定

$$A(\theta) = -\alpha_0(\theta) \tag{13.9.15}$$

对于立方情况

$$\begin{aligned} A(\theta) &= \frac{(-6+11\theta^2)+(6+\theta^2)\cos 2\theta}{6\theta^4} - i\,\mathrm{Im}[\alpha_0(\theta)] \\ &\approx \frac{1}{3} + \frac{1}{45}\theta^2 - \frac{8}{945}\theta^4 + \frac{11}{14175}\theta^6 - i\,\mathrm{Im}[\alpha_0(\theta)] \end{aligned} \tag{13.9.16}$$

当计算光滑函数的傅里叶系数时，自然会得到象 $W(\theta)$ 这样的因子。但是，为了得到精确的积分值，端点校正同样很重要。纳拉西蒙汉(Narasimhan)和卡思基扬(Karthikeyan)[2]给出了一个等式，从代数形式上看等价于我们的梯形公式。但是，他们的公式要求两个 FFT 估算，这是不必要的。这里采用的基本思想可以追回到费龙(Filon)[3]在1928年发表文章(在 FFT 之前！)，他用了辛卜生(Simpson)规则(二次插值)。因为这种插值不是左右对称的，需要两个傅里叶变换。关于等式(13.9.4)的另一个算法由莱尼思(Lyness)在文献[4]中给出；还可以参看文献[5]。据我们所知，我们导出的立方阶公式在以前的文献中未出现过。

当积分范围是$(-\infty,\infty)$时，计算傅里叶变换将有一点让人头疼。如果积分函数在无穷远处很快衰减，那么可以在足够大的 $t$ 值处分段积分。例如对到$+\infty$的积分可以写为

$$\begin{aligned} \int_a^\infty e^{i\omega t}h(t)dt &= \int_a^b e^{i\omega t}h(t)dt + \int_b^\infty e^{i\omega t}h(t)dt \\ &= \int_a^b e^{i\omega t}h(t)dt - \frac{h(b)e^{i\omega b}}{i\omega} + \frac{h'(b)e^{i\omega b}}{(i\omega)^2} - \ldots \end{aligned} \tag{13.9.17}$$

分段点 $b$ 必须选择得足够大，以使剩下的$(b,\infty)$积分很小。它的渐近展开形式中的系列项是通过分部积分得到的。在$(a,b)$内的积分可以用 dfint 计算。在计算容易的情况下保留尽可能多的渐近展开项。可以参见[6]中的示例。更强有的方法，特别适用于长尾函数的方法，却不用 FFT，在文献[7～9]中有叙述。

## 参考文献和进一步读物：


Stoer，J.，and Bulirsch，R. 1980，*Introduction to Numerical Analysis*(New York：Springer-verlag)，p. 88. [1]

Narasimhan，M.S. and Karthikeyan，M. 1984，*IEEE Transactions on Antennas & Propagation*，vol. 32，pp. 404～408. [2]

Filon，L.N.G. 1928，*Proceedings of the Royal Society of Edinburgh*，vol. 49，pp. 38～47. [3]

Giunta，G. and Murli，A. 1987，*ACM Transactins on Mathematical Software*，vol. 13，pp. 97～107. [4]

Lyness，J.N. 1987，*nt Numerical Integration*，P. Keast and G. Fairweather，eds. (DordrZK(echt：Reidel). [5]

Pantis，G. 1975，*Journal of Computational Physics*，vol. 17，pp. 229～233. [6]

Blakemore，M.，Evans，G.A.，and Hyslop，J. 1976，*Journal of Computational Physics*，vol. 22，PP. 352～376. [7]

Lyness，J.N.，and Kaper，T.J. 1987，*SIAM Journal on Scientific and Statistical Computing*，vol. 8，PP. 1005～1011. [8]

Thakkar，A.J.，and Smith，V.H. 1975，*Computer Physics Communications*，vol. 10，pp. 73～79. [9]

## 13.10 小波变换

象快速傅里叶变换(FFT)一样,离散小波变换是一个快速线性运算,它对长度是2的幂次的数据向量进行操作,将它变换成同样长度但在数字上不同的向量。和FFT一样,小波变换是可逆的,事实上还是正交的——逆变换,当看做是一个大矩阵时,就是变换的转置。因此FFT和DWT二者都可以看成是在函数空间中的旋转,从输入空间(或时间)域转换到一个不同的域,其中的基函数是单位向量 $\mathbf{e}_i$,或者从连续的极限角度看是狄拉克(Dirac)$\delta$ 函数。对于FFT,这个新域的基函数是我们熟悉的正弦和余弦函数。在小波域中基函数要复杂一些,并且有一个奇特的名字称"母函数"和"小波"。

当然,对于函数空间可能有无数多个基,它们中的大多数几乎没什么意思;使小波基有意思的是,不象正弦余弦函数,单个的小波函数空间是非常局域化的;同时,象正弦和余弦函数一样,单个小波函数在频率或者(更精确地说)特征尺度上也是非常局域化的。如同在下面我们将会看到的,小波的这种双重局域性使得很多种函数和算符变换到小波域时变得稀疏,或者稀疏到一定的精确度。和傅里叶域相比,一些运算,象卷积等,变得计算很快;同样有一大种类的运算——它可以利用稀疏性的运算——在小波域里计算变得也很快[1]。

不象正弦和余弦,定义了独一无二的傅里叶变换,这里没有一个独特的小波集;实际上可能有无穷多个这样的函数组。粗略地说,不同组的小波在两个方面——空间上局域的疏密程度和光滑程度——进行权衡(还有一些更细微的区别)。

### 13.10.1 德比契斯小波的滤波系数

一组特定的小波由一组特定的数集,称为**小波滤波系数**标定。这里,我们将我们的讨论限制在由德比契斯(Daubechies)发明的一种滤波器。这种滤波器包括的范围从高度局域到高度光滑。最简单的(也最局域的)一类,通常称为 $DAUB4$,仅有四个系数:$c_0,\ldots,c_3$。我们特别对这种滤波器进行讨论。

考虑下面的变换矩阵,将它作用于一列数据向量的左边。

$$\begin{bmatrix}
c_0 & c_1 & c_2 & c_3 & & & & & & \\
c_3 & -c_2 & c_1 & -c_0 & & & & & & \\
 & & c_0 & c_1 & c_2 & c_3 & & & & \\
 & & c_3 & -c_2 & c_1 & -c_0 & & & & \\
\vdots & \vdots & & & & & \ddots & & & \\
 & & & & & & c_0 & c_1 & c_2 & c_3 \\
 & & & & & & c_3 & -c_2 & c_1 & -c_0 \\
c_2 & c_3 & & & & & & & c_0 & c_1 \\
c_1 & -c_0 & & & & & & & c_3 & -c_2
\end{bmatrix} \tag{13.10.1}$$

这里空处表示零。注意此矩阵的结构:第一行产生一个数据与滤波系数 $c_0,\ldots,c_3$ 卷积的分量。与此类似,对第三、第五行和其余奇数行的结果一样。如果偶数行以这种形式出现,正负交替,那么矩阵将是循环的,也就是一般的卷积,可以用FFT方法计算(注意最后两行象具有周期边界条件下的卷积那样环绕起来)。但是偶数行并不是以系数 $c_0,\ldots,c_3$,而是以系数 $c_3,-c_2,c_1,-c_0$ 进行卷积。整个矩阵的作用就是进行两个相关的卷积,然后各去掉一半数值,将剩下的各一半溶在一起。

将滤波器 $c_0,\ldots,c_3$ 看成是一个光滑滤波器，称它为 $H$，$H$ 有点象四个点的移动平均。那么因为负号，滤波器 $c_3,-c_2,c_1,-c_0$，称为 $G$，不是一个光滑滤波器。（在信号处理的文献中，$H$ 和 $G$ 被称为**求积镜象滤波器**[3]。）实际上，$c$ 值的选取是使得 $G$ 对足够光滑的数据向量 $G$ 尽可能产生零响应。这可以通过要求序列 $c_3,-c_2,c_1,-c_0$ 有一定数目的没影矩来完成。当这是 $p$ 矩的情形（从零开始）时，我们说这组小波满足"阶数为 $p$ 的近似条件"。这导致 $H$ 的输出在去掉一半以后，精确地代表了数据的"光滑"信息。$G$ 的输出，同样去掉一半后，被称为数据的"细节"信息[4]。

要使这种特征有用，必须能从它的 $N/2$ 个光滑或 $s$-分量和它的 $N/2$ 个细节或 $d$-分量重建长度为 $N$ 的原始数据向量。这就要求矩阵(13.10.1)是正交的，以使它的逆矩阵就是转置矩阵

$$\begin{bmatrix}
c_0 & c_3 & & & \cdots & & & & c_2 & c_1 \\
c_1 & -c_2 & & & \cdots & & & & c_3 & -c_0 \\
c_2 & c_1 & c_0 & c_3 & & & & & & \\
c_3 & -c_0 & c_1 & -c_2 & & & & & & \\
 & & & & \ddots & & & & & \\
 & & & & c_2 & c_1 & c_0 & c_3 & & \\
 & & & & c_3 & -c_0 & c_1 & -c_2 & & \\
 & & & & & & c_2 & c_1 & c_0 & c_3 \\
 & & & & & & c_3 & -c_0 & c_1 & -c_2
\end{bmatrix} \tag{13.10.2}$$

可以马上看出，要使矩阵(13.10.2)是矩阵(13.10.1)的逆矩阵，当且仅当以下二等式成立

$$\begin{aligned} c_0^2 + c_1^2 + c_2^2 + c_3^2 &= 1 \\ c_2c_0 + c_3c_1 &= 0 \end{aligned} \tag{13.10.3}$$

如果我们附加要求阶数为 $p=2$ 的近似条件，则还有另外两个条件，

$$\begin{aligned} c_3 - c_2 + c_1 - c_0 &= 0 \\ 0c_3 - 1c_2 + 2c_1 - 3c_0 &= 0 \end{aligned} \tag{13.10.4}$$

等式(13.10.3)和(13.10.4)总共有 4 个等式和 4 个未知量 $c_0,\ldots,c_3$，这 4 个等式第一次被德比契斯建立和求解。它的特解（能进行左右逆转）是

$$\begin{aligned} c_0 &= (1+\sqrt{3})/4\sqrt{2} & c_1 &= (3+\sqrt{3})/4\sqrt{2} \\ c_2 &= (3-\sqrt{3})/4\sqrt{2} & c_3 &= (1-\sqrt{3})/4\sqrt{2} \end{aligned} \tag{13.10.5}$$

事实上，DAUB4 是小波组中唯一的最紧密的序列：如果我们有 6 个系数而不是 4 个系数，那么在方程(13.10.3)中需要三个正交条件（偏置值分别为 0、2 和 4），在等式(13.10.4)中我们则需要 $p=3$ 的没影矩。在这种情况下解得的系数，DAUB6也可以用非常紧凑的形式表示：

$$\begin{aligned} c_0 &= (1+\sqrt{10}+\sqrt{5+2\sqrt{10}})/16\sqrt{2} & c_1 &= (5+\sqrt{10}+3\sqrt{5+2\sqrt{10}})/16\sqrt{2} \\ c_2 &= (10-2\sqrt{10}+2\sqrt{5+2\sqrt{10}})/16\sqrt{2} & c_3 &= (10-2\sqrt{10}-2\sqrt{5+2\sqrt{10}})/16\sqrt{2} \\ c_4 &= (5+\sqrt{10}-3\sqrt{5+2\sqrt{10}})/16\sqrt{2} & c_5 &= (1+\sqrt{10}-\sqrt{5+2\sqrt{10}})/16\sqrt{2} \end{aligned} \tag{13.10.6}$$

对更高的 $p$，上至 10，德比契斯(Daubechies)[3]已经列表给出这些数字系数。$p$ 每增加 1，系

数的数目将增加 2 倍。

### 13.10.2 离散小波变换

迄今为止，我们还没有定义离散小变换(DWT)，但是我们几乎到了这一步：DWT 可由分级地应用象等式(13.10.1)的小波系数矩阵而得到，首先是长度为 $N$ 的全数据向量，接着是长度的 $N/2$ 的"光滑"向量，再接着是长度为 $N/4$ 的"光滑——光滑"向量，这样一直进行下云，直到一个数目非常小的(通常是 2 )的"光滑—...—光滑"分量被留下来。这个过程有时被称为"**角锥形算法**"[4]，这样命名的原因是很明显的。DWT 的输出是由留下的分量和所有的"细节"分量构成，"细节"分量是在这个过程中逐步积累起来的。一个方框图可用来清楚地这个过程：

$$
\begin{bmatrix} y_1\\ y_2\\ y_3\\ y_4\\ y_5\\ y_6\\ y_7\\ y_8\\ y_9\\ y_{10}\\ y_{11}\\ y_{12}\\ y_{13}\\ y_{14}\\ y_{15}\\ y_{16} \end{bmatrix}
\xrightarrow{(13.10.1)}
\begin{bmatrix} s_1\\ d_1\\ s_2\\ d_2\\ s_3\\ d_3\\ s_4\\ d_4\\ s_5\\ d_5\\ s_6\\ d_6\\ s_7\\ d_7\\ s_8\\ d_8 \end{bmatrix}
\xrightarrow{\text{排列}}
\begin{bmatrix} s_1\\ s_2\\ s_3\\ s_4\\ s_5\\ s_6\\ s_7\\ s_8\\ \hline d_1\\ d_2\\ d_3\\ d_4\\ d_5\\ d_6\\ d_7\\ d_8 \end{bmatrix}
\xrightarrow{(13.10.1)}
\begin{bmatrix} S_1\\ D_1\\ S_2\\ D_2\\ S_3\\ D_3\\ S_4\\ D_4\\ \hline d_1\\ d_2\\ d_3\\ d_4\\ d_5\\ d_6\\ d_7\\ d_8 \end{bmatrix}
\xrightarrow{\text{排列}}
\begin{bmatrix} S_1\\ S_2\\ S_3\\ S_4\\ \hline D_1\\ D_2\\ D_3\\ D_4\\ \hline d_1\\ d_2\\ d_3\\ d_4\\ d_5\\ d_6\\ d_7\\ d_8 \end{bmatrix}
\xrightarrow{\text{类似进行}}
\begin{bmatrix} S_1\\ S_2\\ \hline \mathscr{D}_1\\ \mathscr{D}_2\\ \hline D_1\\ D_2\\ D_3\\ D_4\\ \hline d_1\\ d_2\\ d_3\\ d_4\\ d_5\\ d_6\\ d_7\\ d_8 \end{bmatrix}
\qquad (13.10.7)
$$

如果数据向量长度是2的更高幂次，那么将有更多步运用等式(13.10.1)(或其它的小波系数)及重新排列。终止点永远是由两个 $s$，分级的 $\mathscr{D}$、$D$ 和 $d$ 构成。注意，$d$ 一旦产生，将简单地沿以后各步传递。

任何一步的 $d_i$ 被称为原始数据向量的"小波系数"；最后的 $S_1$、$S_2$应该严格地称为"母函数系数"，尽管最后的 $d$ 和 $S$ 通常不严格地都称为"小波系数"。因为整个过程由正交的线性运算构成，所以整个 DWT 本身是一个正交的线性运算算符。

为了对 DWT 求逆，只须简单地颠倒整个过程，从最低阶开始，按照等式(13.10.7)从右向左进行。逆矩阵(13.10.2)当然被用来代替矩阵(13.10.1)。

正如已经注意到的，矩阵(13.10.1)和(13.10.2)体现了数据向量的周期性边条件("环绕")。人们通常把这作为一个小的不便接受了：在每一步的最后几个小波系数受到数据向量两端的数据的影响。通过循环移动矩阵(13.10.1)的 $N/2$ 列到左边，就可以将环绕对称化，但是这并不能消除它。实际上有可能通过改变式(13.10.1)中最前 $N$ 行和最后 $N$ 行的系数而完全去掉环绕，即给出一个纯带状对角的正交矩阵。这种变化在有些时候，例如数据向量一端数据的幅度是另一端的很多倍时特别有用，但这已超出我们的范围。

这里的程序，**wt1**，对数据向量 a[1..n] 实施角锥形算法(或它的逆算法，如果 isign 为负)。连续的小波滤波的应用和重新排列由假定的程序 **wtstep** 来完成，**wtstep** 由用户提供(紧接着在下面我们给出了几个不同的 **wtstep** 程序示例)。

```
void wt1(float a[], unsigned long n, int isign,
    void ( * wtstep) (float [], unsigned long, int ))
    一维离散小波变换。此程序实现角锥算法，将它的小波变换代替 a[1..n](如果 isign=1)，或者执行逆运算(如果
    ising=-1)。注意 n 必须是2的整数幂。在调用本程序的时候提供程序 wtstep，它是基本的小波滤波器，wtstep 的例
    子是 daub4和 pwt(在程序 pwtset 之后)。
{
    unsigned long nn;

    if (n < 4) return;
    if (ising >= 0) {                                              子波变换
        for (nn=n;nn>=4;nn>>=1) (* wtstep)(a,nn,ising);            从最大开始，工作到最小级
    } else{                                                        逆子波变换
        for (nn=4; nn<=n;nn<<=1) ( * wtstep)(a,nn,isign);          从最小级开始工作到最大级
    }
}
```

作为 wtstep 的一个特殊示例，下面是 DAUB4小波程序：

```
#include "nrutil.h"
#define C0 0.4829629131445341
#define C1 0.8365163037378079
#define C2 0.2241438680420134
#define C3 -0.1294095225512604

void daub4(float a[], unsigned long n, int isign)
    将戴比契斯4系数的小波滤波器运用到数据向量 a[1..n](如果 isign=1)或者运用它的转置(如果 sisgn=-1)，分
    级地被程序 wt1和 wtn 使用。
{
    float * wksp;
    unsigned long nh,nh1,i,j;

    wksp=vector(1,n);
    if (n < 4) return;
    nh1=(nh=n >> 1)+1;
    if (isign >= 0) {                                                      应用滤波器
        for (i=1,j=1;j<=n-3;j+=2,i++) {
            wksp[i]=C0 * a[j]+C1 * a[j+1]+C2 * a[j+2]+C3 * a[j+3];
            wksp[i+nh] = C3 * a[j]-C2 * a[j+1]-C1 * a[j+2]-C0 * a[j+3];
        }
        wksp[i]=C0 * a[n-1]+C1 * a[n]+C2 * a[1]+C3 * a[2];
        wksp[i+nh] = C3 * a[n-1]-C2 * a[n]+C1 * a[1]-C0 * a[2];
    } else {                                                               应用转置滤波器
        wksp[1]=C2 * a[nh]+C1 * a[n]+C0 * a[1]+C3 * a[nh1];
        wksp[2] = C3 * a[nh]-C0 * a[n]-C1 * a[1]-C2 * a[nh1];
        for (i=1,j=3;i<nh;i++) {
            wksp[j++]=C2 * a[i]+C1 * a[i+nh]+C0 * a[i+1]+C3 * a[i+nh1];
            wksp[j++] = C3 * a[i]-C0 * a[i+nh]+C1 * a[i+1]-C2 * a[i+nh1];
        }
    }
    for (i=1;i<=n;i++) a[i]=wksp[i];
    free_vector(wksp,1,n);
}
```

对于大的小波系数集，对最后几行或几列的环绕在编程上带来很大不便。一个有效的措施是，在主循环之外作为一个特殊情况处理环绕。这里，我们将采用一个更一般的方式，涉及

到在运行时的一些额外算术运算，而能得到一个满意的结果。下面的程序设置了任意特定的小波系数，它们的值是恰好已知道的。

```
typedef struct {
    int ncof, ioff, joff;
    float *cc, *cr;
} wavefilt;

wavefilt wfilt;                          结构的定义说明

void pwtset(int n)
    pwt 的初始化程序，这里实施系数为 4、12 和 20 的德比契斯小波滤性器，系数的选择由 n 的输入值决定。进一步的
    小波滤波器可以用同样明显的方式包括进去。此程序必须在 pwt 第一次使用之前调用。(对 n=4的情况，程序 daub4
    比 pwt 快很多。)
{
    void nrerror(char erro_text[]);
    int k;
    float sig = -1.0;
    static float c4[5]={0.0,0.4829629131445341,0.8365163037378079,
            0.2241438680420134,-0.1294095225512604};
    static float c12[13]={0.0,0.111540743350,0.494623890398,0.751133908021,
        0.315250351709,-0.226264693965,-0.129766867567,
        0.097501605587,0.027522865530,-0.031582039318,
        0.000553842201,0.004777257511,-0.001077301085};
    static float c20[21]={0.0,0.026670057901,0.188176800078,0.527201188932,
        0.688459039454,0.281172343661,-0.249846424327,
        -0.195946274377,0.127369340336,0.093057364604,
        -0.071394147166,-0.029457536822,0.033212674059,
        0.003606553567,-0.010733175483,0.001395351747,
        0.001992405295,-0.000685856695,-0.000116466855,
        0.000093588670,-0.0000132642203};
    static float c4r[5],c12r[13],c20r[21];

    wfilt.ncof=n;
    if (n == 4) {
        wfilt.cc=c4;
        wfilt.cr=c4r;
    }
    else if (n == 12) {
        wfilt.cc=c12;
        wfilt.cr=c12r;
    }
    else if (n == 20) {
        wfilt.cc=c20;
        wfilt.cr=c20r;
    }
    else nrerror("unimplemented value n in pwtset");
    for (k=1;k<=n;k++) {
        wfilt.cr[wfilt.ncof+1-k]=sig * wfilt.cc[k];
        sig = -sig;
    }
    wfilt.ioff = wfilt.joff = -(n >> 1);   这些值在每级以小波的"支柱"为中心分布。小波的"峰"可以通过
                                           选择 ioff=-2和 joff=-n+2来近似。
}
```

一旦 pwtset 被调用，下面的程序可作为 wtstep 的一个特例：

```
#include "nrutil
```

```
typedef struct {
    int ncof, ioff, joff;
    float *cc, *cr;
}wavefilt;

extern wavefilt wfilt;                    由 pwtset 定义

void pwt(float a[], unsigned long n, int isign)
```

部分小波变换:将一个任意的小波滤波器应用到数据量 a[1..n](对 isign=1情形)或使用它的转置(对 isign=−1情形)。此程序被 wt1和 wtn 分级调用.实际的滤波器,在进行滤波之前,(必须的)调用 pwtset 来确定,它是对结构 wfilt 作初始化。

```
{
    float ai,ai1,*wksp;
    unsigned long i,ii,j,jf,jr,k,n1,ni,nj,nh,nmod;

    if (n < 4) return;
    wksp=vector(1,n);
    nmod=wfilt.ncof*n;                    对模 n 等于零的正常数
    n1=n-1;                               因为 n 是 2 的幂次, 所以屏蔽所有位
    nh=n >> 1;
    for (j=1;j<=n;j++) wksp[j]=0.0;
    if (isign >= 0) {                     求滤波
        for (ii=1,i=1;i<=n;i+=2,ii++) {
            ni=i+nmod+wfilt.ioff;         指针增加并环绕
            nj=i+nmod+wfilt.joff;
            for (k=1;k<=wfilt.ncof;k++) {
                jf=n1 & (ni+k);           用逐位并且环绕指针
                jr=n1 & (nj+k);
                wksp[ii] += wfilt.cc[k]*a[jf+1];
                wksp[ii+nh] += wfilt.cr[k]*a[jr+1];
            }
        }
    } else {                              求转置滤波
        for (ii=1,i=1;i<=n;i+=2,ii++) {
            ai=a[ii];
            ai1=a[ii+nh];
            ni=i+nmod+wfilt.ioff;         看上面的说明
            nj=i+nmod+wfilt.joff;
            for (k=1;k<=wfilt.ncof;k++) {
                jf=(n1 & (ni+k))+1;
                jr=(n1 & (nj+k))+1;
                wksp[jf] += wfilt.cc[k]*ai;
                wksp[jr] += wfilt.cr[k]*ai1;
            }
        }
    }
    for (j=1;j<=n;j++) a[j]=wksp[j];      从工作空间复制回结果
    free_vector(wksp,1,n);
}
```

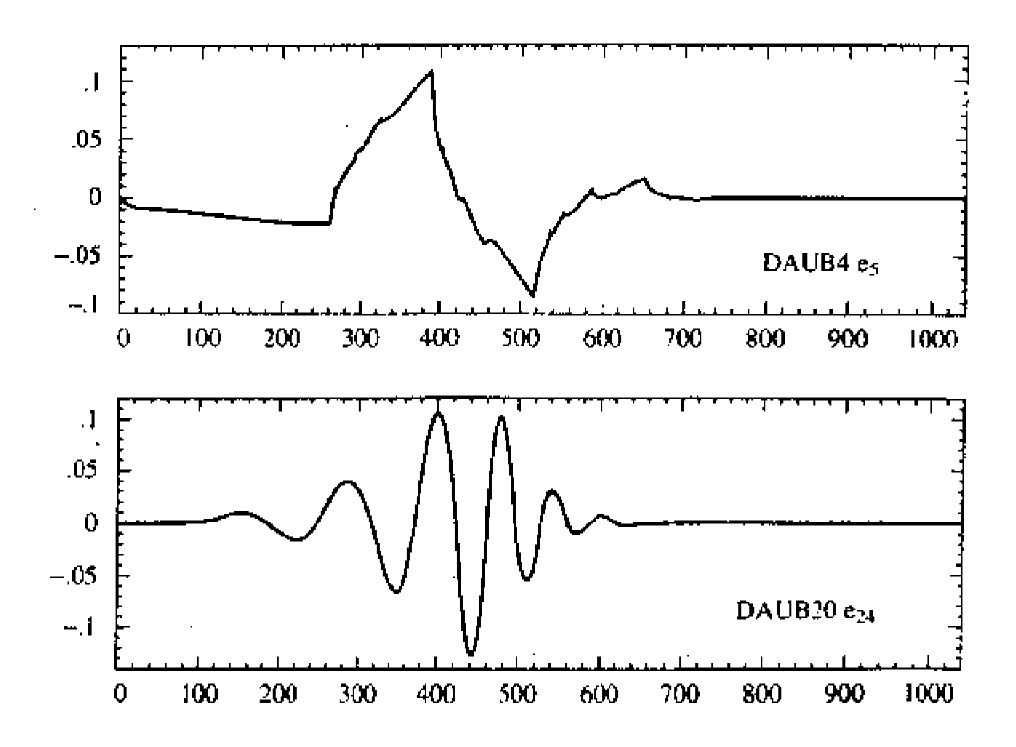
### 13.10.3 小波象什么

现在我们来看一些实际小波的情况。为此,我们通过上面的离散小波变换简单地运行于单位向量,并且再使 isign 为负以进行逆变换。图13.10.1显示了 DAUB4 小波,它是长度为1024 的向量的第 5 个分量构成的单位向量的逆 DWT,以及 DAVB20 小波,它是第 24 个分量的逆 DWT(人们必须选择 DAVB20 后面级别的小波,以避免带有环绕尾巴的小波)。其它的单位向量给出相同的形状的小波但有不同的位置和标度。

可以看出 DAUB4 和 DAUB20 有连续的小波。DAUB20 还有更高阶的连续导数。

DAUB4 有一个特别的性质，它的导数几乎存在于各处。不存在的地方则在 $p/2^n$ 点处，其中 $p$ 和 $n$ 是整数；在这些点处，DAUB4 左边可微，而右边不可微。这种不连续性——至少在某些导数处——是带有紧密支柱的小波的必然特征，如德比契斯数列。小波系数的数目每增加 2，德比契斯小波获得一半的连续导数。(但并不是精确的一半；精确的数目是一个无理数！)

注意如下的事实，小波不光滑并不能阻止它们精确代表一些光滑函数，如同它们的近似阶数 $p$ 要求的那样。小波的连续性和由一系列小波代表的函数的连续性并不一样。例如，DAUB4 可以代表(分段)任意斜率的线性函数：通过适当的线性组合，所有的尖点被消除。系数数目每增加 2，容许高1阶的多项式被精确代表。




小波族 DAVB4和 DAVB20中的单个基函数。一个完整的正交小波基包括这些函数的任一种平移和标度。DAVB4有无穷多个尖点；DAVB20显示了有高阶导数相似的行为。

图13.10.1 小波函数

图13.10.2显示了对输入向量 $e_{10}+e_{58}$，又是两个不同的特别小波，实施逆 DWT 所得的结果。因为 10 在等级范围 9～16 中处于较早的位置，所以对应的小波出现在图形的左边；因为 58 处在等级范围较后的位置(较小尺度)，所以它是一个较窄的小波，而且处在范围 33～64 内，它接近末端，因此波形处在图形的右边。注意小尺寸的小波幅度更高，因此这两个小波有相同的平方积分。

### 13.10.4 傅里叶域中的小波滤波

傅里叶变换的一组滤波系数 $c_j$ 由下式确定：

$$H(\omega) = \sum_j c_j e^{ij\omega} \qquad (13.10.8)$$

这里 $H$ 是周期为 $2\pi$ 的周期函数，它和以前有相同的意义：它是小波滤波器，现在在傅里叶域中重新写出。一个非常有用的事实是，$c$ 的正交条件(参看上面和等式(13.10.3))在傅里

叶域中变成了两个简单的关系：

$$\frac{1}{2}|H(0)|^2 = 1 \qquad (13.0.9)$$

和

$$\frac{1}{2}[|H(\omega)|^2 + |H(\omega+\pi)|^2] = 1 \qquad (13.10.10)$$

类似地，$p$ 阶的近似条件（参见前面的等式(13.10.3)）有一个简单的公式，要求 $H(\omega)$ 在 $\omega=\pi$ 处有一个第 $p$ 阶零点，或写为

$$H^{(m)}(\pi) = 0 \qquad m = 0,1,\ldots,p-1 \qquad (13.10.11)$$

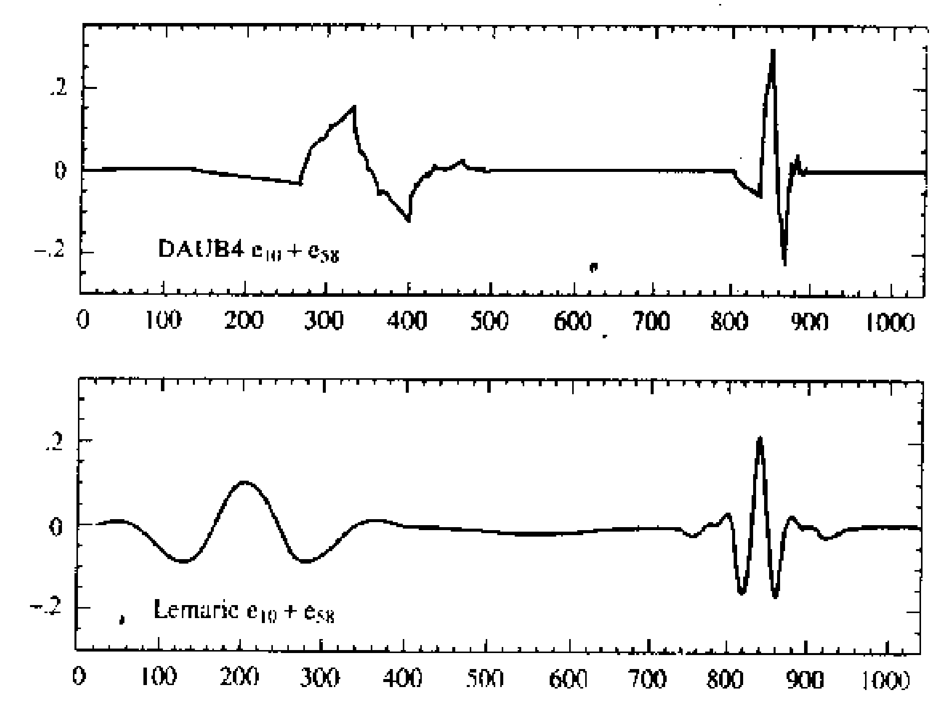
因此在傅里叶域里构造小波系列就比较直观。只要简单地构造一个满足等式(13.10.9)～(13.10.11)的函数 $H(\omega)$。为了得到实际的 $c_j$，它可应用于长度为 N 的数据向量（或 $s$-分量），并且又具有如同矩阵(13.10.1)和(13.10.2)的周期环绕，因此就可以利用离散傅里叶变换得到等式(13.10.1)和(13.10.2)的周期环绕，利用离散傅里叶变换得到等式(13.10.8)的逆等式

$$c_j = \frac{1}{N}\sum_{k=0}^{N-1} H\left(\frac{2\pi k}{N}\right)e^{-2\pi ijk/N} \qquad (13.10.12)$$

求积镜象滤波器 $G$（将 $G_j$ 倒序并符号交替变换）恰好有傅里叶表达式

$$G(\omega) = e^{-i\omega}H^*(\omega+\pi) \qquad (13.10.13)$$

其中的星号代表复共轭。




图中小波是由两个单位向量，$e_{10}+e_{58}$ 之和产生的，它们有不同的标度等级，也在不同的空间位置，DAUB4小波(a)是由在坐标空间的滤波器（等式(13.10.5)）定义的，而莱莫瑞小波(b)是由在傅里叶空间中很容易写出的滤波器（等式(13.10.14)）定义的。

图13.10.2　更多的小波示例

一般说来，上面的过程将不会产生具有紧密柱的小波滤波器。换言之，$N$ 个 $c_j(j=$

0,...,N-1)一般是非零值(尽管它们在幅度上可能会很快减小)。德比契斯小波,或其它的有紧密支柱小波都是特别选择,以使得 $H(\omega)$ 是一个仅有少数几个傅里叶分量的三角多项式,以保证只有少数几个非零的 $c_j$。

另一方面,有时并没有特别的原因要求紧密的支柱。放弃这个要求将容许构造更光滑的小波(更高的 $p$ 值)。即使没有紧密支柱,在矩阵(13.10.1)中隐含的卷积均可以用 FFT 方法有效地计算。

莱莫瑞(Lemarie)小波(参看文献[4])有 $p=4$,但没有紧密支柱,并且通过选择 $H(\omega)$ 来定义:

$$H(\omega)=\left[2(1-u)^4\frac{315-420u+126u^2-4u^3}{315-420v+126v^2-4v^3}\right]^{1/2} \qquad (13.10.14)$$

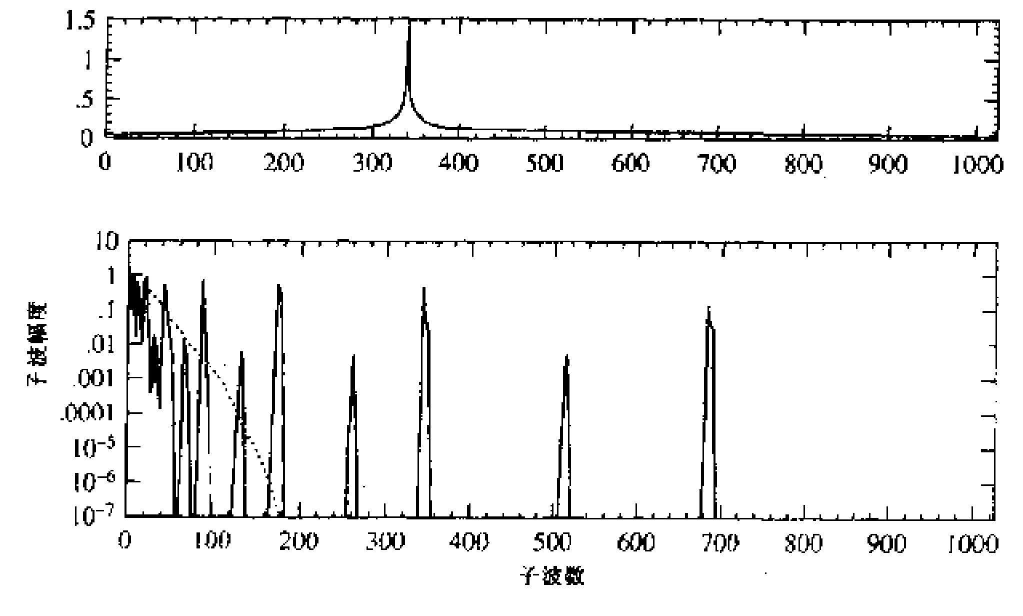
其中

$$u\equiv\sin^2\frac{\omega}{2} \qquad v\equiv\sin^2\omega \qquad (13.10.15)$$

解释等式(13.10.14)来源超出了我们范围。一个不正规的描述是:由等(13.10.14)导出的求积镜象滤波器 $G(\omega)$ 具有性质——当它运用到某种函数,这种函数的奇数样本值等于它偶数样本值的立方样条插值时,它给出恒等于0的结果。因为这类函数包括很多光滑的成员,因此 $H(\omega)$ 选择一个函数的光滑信息内容时做得非常好。莱莫瑞小波示例显示在图13.10.2 中。

### 13.10.5 被截小波近似

小波有用性的一个重要方面在于这样一个事实:小波变换可以被严重截去而依然有用;也说是说对小波可以进行稀疏展开。傅里叶变换的情形就不同:FFT 通常是没有被截短地运用,例如,进行快速卷积就是如此。之所以能进行这种运算是因为卷积算法在傅里叶基中特别简单。但是在小波基中,没有任何标准的算术运算是特别简单的。




(a)带有尖顶的任意检验函数,取样成为度为1024的向量;(b)对(a)进行离散小波变换后产生的1024个小波系数的绝对值。注意图中的对数坐标。当按大小降低的顺序排序时,图中点曲线线具有同样的幅度,可以看出1024个系数中仅130个系数的值大于$10^{-4}$(或大于$10^{-5}$乘以最大系数值,这个系数值约为10)

图13.10.3

为了明白截除是怎样起作用的，考虑在上页图13.10.3 中的简单示例。上面的方框中列出一个任意被选取的检验函数，除了一个平方根的尖点外是光滑的，它被取样成长度为 $2^{10}$ 的向量。下面的方框（实线）用对数的刻度显示了向量经过 DAVB4离散小波变换后的分量的绝对值。注意从右到左是不同的谱级别，513～1024，257～512，129～256 等 在每 级，小波系数仅在紧靠近尖峰的地方，或者紧靠谱范围的左边界和右边界（边缘效应）才不可忽略。

在上页图13.10.3的下方框图中的点线和实线具有相同的幅度，但点曲线是按系数值的大小降低的顺序排列。例如，从图上马上可以看出第 130 个最大的小波系数的幅度值比小波的最大幅度的$10^{-5}$倍还要小，小波系数最大幅度值接近～10（功率或平方积之比小于$10^{-10}$）。因此，作为示例的函数可以仅被 130 个系数精确代表，而不是 1024 个系数——余下系数可令其为 0 。可以看出这种截除使向量变稀疏，但不差于 1024 。还有一点很重要，那就是在小波空间中向量的截除应根据分量的幅度，而不是各分量在向量中的位置。但是保留向量的前 256 个分量（这向量占有除了最后两个级的所有谱系）将对函数给出一个非常差的、边缘参差不齐的近似。因此当用小波压缩一个函数时，必须同时记录非零系数的值和位置。

一般来说，紧密（因此不光滑）小波更适合于较低精度的近似和带有不连续（象边缘）的函数，而光滑（因此不紧密）的小波则适合获得较高数字精确度。这种性质使得紧密小波成为图像压缩的较佳选择，光滑小波成为积分等式快速计算的最好选择。

### 13.10.6 多维小波变换

一个 $d$ 维数组的小波变换很容易根据其下标顺序地变换而得到，首先是第一个下标（对其它下标的所有值），接着是第二个下标，等等。每个变换对应于乘以一个正交矩阵。由于矩阵的结合性，所以其结果和下标被变换的顺序无关。此情形和多维的 FFT 非常类似。因此有效地进行多维 DWT 的程序可以用多维的 FFT 程序如 **fourn** 为模型而建立起来。

```
#include "nrutil.h"

void wtn(float a[], unsigned long nn[], int ndim, int isign,
    void (*wtstep)(flaot [], unsigned long, int))
    如果isign 输入量为1，那么将用 a 的 ndim 维离散小波变换替换 a。nn[1..ndim]是一个整数数组，其值为每维的长度
    （实值数），并且还必须是2的幂次。a 是一个实数数组，其长度为上面各个长度的积，在 a 中数据以多维实型数组的形
    式储存。如果 isign 以－1输入，那么 a 将被它的逆小波变换替换。程序 wtstep 是基本的小波滤波器，调用此程序之前
    它必须提供。wtstep 的例子有 daub4和 pwt（必须将 pwtset 包括在前）。
{
    unsigned long i1,i2,i3,k,n,nnew,nprev=1,nt,ntot=1;
    int idim;
    float *wksp;

    for (idim=1;idim<=ndim;idim++) ntot *= nn[idim];
    wksp=vector(1,ntot);
    for (idim=1;idim<=ndim;idim++) {                    对维数的主循环
        n=nn[idim];
        nnew=n*nprev;
        if (n > 4) {
            for (i2=0;i2<ntot;i2+=nnew) {
                for (i1=1;i1<=nprev;i1++) {
                    for (i3=i1+i2,k=1;k<=n;k++,i3+=nprev) wksp[k]=a[i3];   复制有关行
                                                                           或列到工作
```

```
                if (isign >= 0) {                                         空间
                    for(nt=n;nt>=4;nt >>= 1)                              进行一维子波变换
                        (*wtstep)(wksp,nt,isign);
                } else {                                                  或逆变换
                    for(nt=4;nt<=n;nt <<= 1)
                        (*wtstep)(wksp,nt,isign);
                }
                for (i3=i1+i2,k=1;k<=n;k++,i3+=nprev) a[i3]=wksp[k];     从工作空间
                                                                          拷回
            }
        }
    }
    nprev=nnew;
  }
  free_vector(wksp,1,ntot);
}
```

这里和以前一样，**wtstep** 是单独的小波滤波器，可以是 **daub4** 或 **pwt**。

## 13.10.7 图像压缩

多维变换 wtn 的一个直接应用是图像压缩。整个过程就是数字化的图像进行小波变换，接着用一种非均匀的、优化的方法在小波系数中"分配二进制位"。一般说来，大的小波系数得到精确的量化，而小的系数量化很粗糙，仅有一两位二进位——或者完全截除了。如果结果的量化等级从统计上看还不均匀，它们还可以用一些技巧例如霍夫曼(Huffman)编码(第20.4节)进一步压缩。

尽管这个过程"后部分"的更细微的描述，也就是图像的量化和编码，超出了我们的范围，但是对进行简单截除的"前部分"小波编码的说明还是很直观的：我们保留大于某个阈值的所有小波系数(保持充分的精确性)，删除(置为 0 )所有较小的小波系数。我们可以通过调整阈值而改变被保留系数所占的百分比。

图13.10.4显示了保留不同数目的小波系数的一系列图像。原始图像(a)，一幅 IEEE 官方的测试图象，有 256×256 个象素点以及 8 个二进位灰度级。紧接着的两个重建图像是由 65536 个小波系数的 23%(b)和 5.5%(c)构造的。后一幅图象显示了截去小波系数后的一种折衷。高对比度边缘(模特儿的右脸颊和头发的亮光等)依然保持相对较高的分辨率，而低对比度区域(模特儿的左眼和左颊等)则退化为相当于原始象素点的分辨率。图13.10.4(d)是实施傅里叶变换而不是小波变换的等价结果：图像是由 65536 个实傅里叶分量中具有最大幅度 5.5%个分量重新构造的。可以看出，由于正弦和余弦不是局域的，整个图像的分辨率普遍比较差。任何分量的删除处产生模糊的"边缘"。(因此，实际的傅里叶图像压缩方式是将图像拆成小的象素块，例如 16×16 ，图象重建时，再对每块的边缘做相当精细的光滑处理。)

## 13.10.8 线性系统的快速求解

小波的一个最有趣的潜在的应用是线性代数的应用。其基本思想[1]是，将整数运算算子(也就是一个大矩阵)看成一幅数字图像。假设算子在二维小波变换下能很好地压缩，即它的小波系数大部分非常小以致可忽略。那么，与算子有关的任何线性系统在小波基下成为一个稀疏系统，换言之，为了求解

$$\mathbf{A}\cdot\mathbf{x}=\mathbf{b} \tag{13.10.16}$$

我们首先对算符 **A** 和右边的 **b** 进行小波变换：

$$\widetilde{\mathbf{A}}\equiv\mathbf{W}\cdot\mathbf{A}\cdot\mathbf{W}^T,\quad \widetilde{\mathbf{b}}\equiv\mathbf{W}\cdot\mathbf{b} \tag{13.10.17}$$

其中 **W** 代表了二维小波变换，接着求解

$$\widetilde{\mathbf{A}}\cdot\widetilde{\mathbf{x}}=\widetilde{\mathbf{b}} \tag{13.10.18}$$

最后通过逆小波变换得到结果

$$\mathbf{x}=\mathbf{W}^T\cdot\widetilde{\mathbf{x}} \tag{13.10.19}$$

（注意，程序 **wtn** 完成了将 **A** 到 $\widetilde{\mathbf{A}}$ 的完全变换）。



(a)IEEE的测试图像，256×256个象素点，8位灰度级(b)将图像变换成小波基，其小波分量的77%被置为零(那些振幅最小的)；接着从剩下的23%重建图像。(c)和(b)一样，但是94.5%的小波分量被删除。(d)与(c)一样，但是用傅里叶变换代替了小波变换。小波系数在保留有关细节方面优于傅里叶系数。

图13.10.4

一个典型的能很好地进行小波压缩的矩阵算子是，在靠近它的主对角线有任意的（甚至

是奇异的)元素，但在这远离对角线处将变光滑

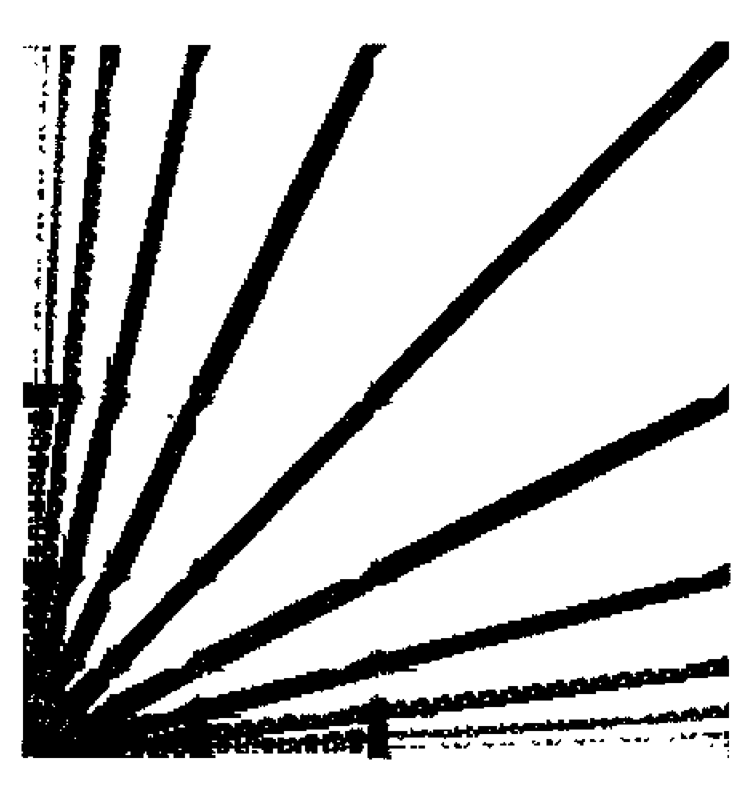
$$A_{ij} = \begin{cases} -1 & 若\ i = j \\ |i-j|^{-1/2} & 其它情况 \end{cases} \qquad (13.10.2)$$

图13.10.5显示了这个矩阵小波变换的图形形式，其中 $i$ 和 $j$ 的变化范围是 1...256。采用 DAUB12小波，将幅度大于最大元素幅度$10^{-3}$倍的元素，用黑的象素点表示，在$10^{-3}$和$10^{-6}$之间的元素用灰的象素点表示，白的象素点表示元素幅度$<10^{-6}$。下标 $i$ 和 $j$ 都从最低最左边计起。

从图中可以看出，谱系分解成大小为 2 的幂次方的块。在每个块的边缘或角落，可以看到由于环绕的子波边界条件引起的边缘效应。除了边缘效应外，在每一块之内，不可忽略的元素是集中在块的对角线上。这说明，对于这种线性算符，一个小波主要将和它自己谱系的近邻(沿着主对角线的方块)和其它谱系的近邻(对角线之外的矩形块)联系在一起。

不可忽略的矩阵元素的个数，如图13.10.5所示，数量级仅为 $N$，即矩阵的线性尺寸；粗略地估计，其数目为 $10N\log_{10}(1/\epsilon)$，其中 $\epsilon$ 为截断水平，例如 $10^{-6}$。对于一个 $2000\times2000$ 的矩阵，那么矩阵的稀疏因子将接近 30 。

有各种数值方法可以用来求解这种"谱系对角带"形式的稀疏线性系统。比尔琴(Beulkin)、克夫曼(Coifman)和罗克林(Rokhlin)[1]进行了有趣的观察得到：(1)两个这种矩阵的乘积还是谱系对角带矩阵(当然还是截除了新产生的元素中小于预先确定的阈值 $\epsilon$)；而且还有(2)乘积可用 $N$ 次操作顺序运算而形成。



原始矩阵在它的对角线上有不连续的尖顶，而到对角线两边逐渐光滑地衰减。在小波基下，矩阵变得稀疏：大于$10^{-3}$的分量以黑色表示，大于$10^{-6}$的分量以灰色表示，小于$10^{-6}$的分量以白色表示。矩阵下标 $i$ 和 $j$ 从最下端左边计起。

图13.10.5　用图表示一个$256\times256$矩阵的小波变换

由舒尔兹(Schultz)或霍特林(Hotelling)方法,快速的矩阵相乘法使得人们能对矩阵求逆,参看第2.5节。

对于谱系对角带的快速计算还有其它的方式。例如,可以采用第2.7节中 **linbcg** 共轭梯度方法。

参考文献和进一步读物:


Daubechies, I. 1992, *Wavelets* (Philadelphia: S. I. A. M).

Strang, G. 1989, *SIAM Review*, vol, 31. pp. 614~627.

Beylkin, G. Coifman, R., and Rokhlin, V. 1991, *Communications on Pure and Applied Mathematics*, vol. 44, pp. 141~183. [1]

Daubechies, I. 1988, *Communications on Pure and Applied Mathematics*, vol. 41, pp. 909~996. [2]

Vaidyanathan, P. P. 1990, *Proceedings of the IEEE*, vol. 78, pp. 56~93. [3]

Mallat, S. G. 1989, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, pp. 674~693. [4]

Freedman, M. H., and Press, W. H. 1992, preprint. [5]


## 13.11 取样定理的数值应用

在第6.1节由于赖比契(Rybicki)的贡献,我们对道生(Dawson)积分得到了一个近似公式,现在我们有了傅里叶变换的基础。我们可以看出此公式来源于取样定理的数值应用,这时取样定量一般被看做是一个纯粹的分析工具。我们的讨论和赖比契(Rybicki)[1]一样。

对于目前的目的,取样定理表述为如下形式最为方便:考虑一个任意函数 $g(t)$ 和取样点网格 $t_n=\alpha+nh$,其中 $n$ 按整数变化,$\alpha$ 是一个常数,它允许取样网格的任意移动。那么我们有:

$$g(t)=\sum_{n=-\infty}^{\infty}g(t_n)\ \mathrm{sinc}\ \frac{\pi}{h}(t-t_n)+e(t) \tag{13.11.1}$$

其中 $\mathrm{sinc}\ x=\sin x/x$。对取样点的求和被称为 **g(t)的取样表示**,$e(t)$ 是误差项。取样定理证明,如果 $g(t)$ 的傅里叶变换

$$G(\omega)=\int_{-\infty}^{\infty}g(t)e^{i\omega t}dt \tag{13.11.2}$$

在 $|\omega|\geqslant\frac{\pi}{h}$ 时趋于 0,则取样表示是精确的,也就是说 $e(t)=0$。

取样表示在什么时候对函数的近似数值计算最为有利呢?为使得误差项尽可能小,傅里叶变换 $G(\omega)$ 在 $|\omega|\geqslant\frac{\pi}{h}$ 时必须足够小。另一方面,为了使式(13.11.1)中的求和能用合理的、数目较少的项近似,函数 $g(t)$ 在 $t$ 的某一范围之外必须非常小,因此为了使式(13.11.1)数值上有用,我们得到两个条件:函数 $g(t)$ 和它的傅里叶变换 $G(\omega)$ 在它们各自的自变量比较大时应很快趋于零。

遗憾的是,这两个条件是互相对立的——如同量子力学里的不确定性原理。两个自变量以多快的速度趋于零存在严格的限制。根据哈迪(Hardy)[2]定理,如果有当 $|t|\to\infty$,$g(t)=O(e^{-t^2})$ 和当 $|\omega|\to\infty$,$G(\omega)=O(e^{-\omega^2/4})$,那么 $g(t)=Ce^{-t^2}$,其中 $C$ 是一个常数。这可以解释为什么所有的函数中,高斯函数随 $t$ 和 $\omega$ 的衰减都是最快的,从这个意义上讲,高斯函数在数值上是所有函数中能最好地进行取样表示的。

我们将高斯函数写为 $g(t)=e^{-t^2}$,

$$e^{-t^2}=\sum_{n=-\infty}^{\infty}e^{-t_n^2}\mathrm{sinc}\ \frac{\pi}{h}(t-t_n)+e(t) \tag{13.11.3}$$

误差 $e(t)$ 除了依赖于 $t$ 外,还依赖于 $h$ 和 $\alpha$,在目前的情况下,对 $e(t)$ 给定如下的一个范围就足够了,

$$|e(t)|<e^{-(\pi/2h)^2} \tag{13.11.4}$$

它可以简单地理解为高斯函数在分界点 $\omega > \frac{\pi}{h}$ 处的傅里叶变换之幅度的量级。

当和式(13.11.3)是以有限项近似,例如从 $N_0 - N$ 到 $N_0 + N$,其中 $N_0$ 是最接近 $\frac{a}{h}$ 的整数,将还有进一步的截断误差。但是,如果 N 被选择得使 $N > \pi/(2h)^2$,那么截断误差将小于式(13.11.4)给定的范围。因为这个范围是一个保守估计,我们将继续用它估计式(13.11.3)中的 $\epsilon(t)$。即使在 $N$ 取中等的数值时,被截断的求和式也会对高斯函数给出相当精确的表示,例如当 $h=1/2$,$N=7$ 时,$|\epsilon(t)| < 5\times 10^{-5}$;当 $h=1/3$,$N=15$ 时 $|\epsilon(t)| < 2\times 10^{-10}$;当 $h=1/4$,$N=25$ 时 $|\epsilon(t)| < 7\times 10^{-18}$。

人们可能要问,高斯函数的这样一个数值表示,可以象指数函数一样计算得那样容易和快速,其用途是什么呢?答案在于很多超越函数可表示成与高斯函数有关的积分,通过式(13.11.3)的代换,就可以用基本函数的和很好地近似求积分。

让我们首先考虑复变量为 $z = x + iy$ 的函数 $w(z)$ 的例题,它和复误差函数存在如下关系:

$$w(z) = e^{-z^2}\operatorname{erfc}(-iz) \tag{13.11.5}$$

它的积分表示是

$$w(z) = \frac{1}{\pi i}\int_C \frac{e^{-t^2}dt}{t - z} \tag{13.11.6}$$

其中的积分曲线 C 从 $-\infty$ 延伸到 $\infty$,从 $z$ 轴通过(参看文献[3])。有许多方法可以用来估计这个函数(参看文献[4]),将取样表示式(13.11.3)代入(13.11.6)式,并且进行基本的曲线积分,我们有:

$$w(z) \approx \frac{1}{\pi i}\sum_{n=-\infty}^{\infty} h e^{-t_n^2}\frac{1 - (-1)^n e^{-\pi i(\alpha - z)/h}}{t_n - z} \tag{13.11.7}$$

其中我们忽略了误差项。注意对某些 $n = m$,在 $z \to t_m$ 时没有奇点,但是在这种情况下,对第 $m$ 项必须进行特殊处理(例如,通过幂次数展开)。

等式(13.11.7)的另一种表现形式,可以通过用三角函数表达式(13.11.7)中的复指数,并且利用 $z$ 代替取样表达式(13.11.3)中的 $t$ 而得到:

$$w(z) \approx e^{-z^2} - \frac{1}{\pi i}\sum_{n=-\infty}^{\infty} h e^{-t_n^2}\frac{1 - (-1)^n\cos\pi(\alpha - z)/h}{t_n - z} \tag{13.11.8}$$

此形式在 $|y| \ll 1$ 时,用来求 Re $w(z)$ 特别有用。注意在估算式(13.11.7)时,求和号中的指数项是一个常数,只须估算一次,对式(13.11.8)中的余弦项也类似。

通过选择特定的 $\alpha$ 值,可以从等式(13.11.7)或(13.11.8)导出一组不同的公式。有意义的 8 个值是:$\alpha = 0$、$x$、$iy$ 或 $z$,再加上每个值增加 $h/2$ 后所得的值。因为式(13.11.3)中误差范围表达式已经假设 $\alpha$ 为实值,选择 $\alpha$ 为复数仅当 $z$ 的虚部不是很大时才有用。这里不可能列出 16 个可能等式,我们仅给出能显示一些重要特征的两个特例。

首先在等式(13.11.8)中记 $\alpha = 0$,则有

$$w(z) \approx e^{-z^2} + \frac{1}{\pi i}\sum_{n=-\infty}^{\infty} h e^{-n^2h^2}\frac{1 - (-1)^n\cos(\pi z/h)}{nh - z} \tag{13.11.9}$$

在整个 $z$ 平面此近似式应用得很好。如同我们前面所述,当某一些的分母比较小时,必须采用幂级数展开。$\alpha = 0$ 的公式在文献[5]中有简明的讨论。它们和齐瑞勒(Chiarella)、瑞奇尔(Reichel)[6]用顾德文(Goodwin)[7]方法导出的公式相似,但并不等价。

下一步,在式(13.11.7)中,令 $\alpha = z$,我们有:

$$w(z) \approx e^{-z^2} - \frac{2}{\pi i}\sum_{n\,\text{奇数}}\frac{e^{-(z - nh)^2}}{n} \tag{13.11.10}$$

其中的求和是对所有的奇整数(正的和负的)求和。注意在求和式中我们已经做替换 $n \to -n$。此公式比式(13.11.9)简单,只包含一半的项数,但是当 y 大时,误差将增大。等式(13.11.10)就是在第6.10节中用的道生(Dawson)积分近似公式(6.10.3)的一般化。

## 参考文献和进一步读物：


Rybicki, G. B. 1989, *Computers in Physics*, vol. 3, no. 2, pp. 85~87. [1]

Hardy, G. H. 1933, *Journal of the London Mathematical Society*, vol. 8, pp. 227~231. [2]

Abramowitz, M., and Stegun, I. A. 1964, *Handbook of Mathematical Functions*, *Applied Mathematics Series*, vol. 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York). [3]

Gautschi, W., 1970, *SIAM Journal on Numerical Analysis*, vol. 7, pp. 187~198. [4]

Armstrong, B. H., and Nicholls, R. W. 1972, *Emission, Absorption and Transfer of Radiation in Heated Atmospheres* (New York: Pergamon). [5]

Chiarella, C. and Reichel, A. 1968, *Mathematics of Computation*, vol. 22, pp. 137~143. [6]

Goodwin, E. T. 1949, *Proceedings of the Cambridge Philosophical Society*, vol. 45, pp. 241~245. [7]

# 第十四章　数据的统计描述

## 14.0　引言

在这一章和下一章，讨论数据的概念将占据突出的地位。

数据自然是由数组成的，但这些数是输入到计算机中去的，而不是由计算机自身产生的。这些数必须认真地处理，既不能篡改，也不应对它们进行一些不完全了解特性的数值处理。对数据应有尊重的态度，而不能采取在其它数值游戏中所允许的、甚至受到赞扬的"华而不实"的态度。

数据的分析不可避免地要涉及到某些统计学领域，统计学可以处理一些问题，这些问题既不能确切地说是数学分支的问题，或许也不能说是科学分支的问题。在以下各节，将经常遇到如下范例：

- 为了计算"统计量"，将对数据应用一些公式进行处理；
- 计算这一统计量落在概率分布的那一部分，这些概率分布是基于"零假说"来计算的；
- 如果它落在一个非常不可能的点上，即在概率分布的末端以外，就可以说"零假说"对这数据组来说是错误的。

如果一个统计量落在分布的合理部分，也不能说"零假说"被"验证"或"证明"。这是统计学的独特之处，它从来不能证明任何事物成立，而仅仅只反驳事物！证明一个假说的最好方式是通过排除一系列相互竞争的假说，而这些假说是已被人们一一提出的。一段时间后，对手和竞争者放弃寻求新的假说，那么所做的假说将被接受。这听起来多么古怪，但你知道，这就是科学工作！

在本书中，我们对数据分析过程做了稍稍有点主观的区分，一是独立于模型的，另一是依赖于模型的。前者包含了所谓的**描述性统计学**，用一些常规的量来表征一组数据组，如均值、方差等。它也包括统计检验，寻求建立两个或更多数据组之间的"相同之处"或"不同之处"，或者寻求建立或测度两组数据之间的相关程度，这些内容将在这一章中予以讨论。

在另一类依赖于模型的统计学中，我们将整个拟合数据的主题纳入这种理论，如参数估值、最小二乘方拟合等等，这些内容将在第十五章予以介绍。

第14.1节将处理集中趋势的测度，如分布的矩、中值和众数。在第14.2节我们将学习检验不同的数据组，是否是由这些集中趋势测度的不同值的分布导出。这又自然地引出第14.3节中更一般的问题，两个分布是否能被有效地表明是不同的分布。

从第14.4节至第14.7节我们将讨论关于两种分布之间的关联的测度，我们希望能决定两个变量是否是互相"相关"或"依赖"的。如果是，我们希望能以某种简单的公式表征它们相关的程度。同时还强调了参数和非参数(秩)方法之间的差别。

第14.8节将引入数据平滑的概念，并且讨论了一特例：Savitzky-Golay 平滑滤波器。

本章在数学上，使用了第六章的特殊函数，特别是第6.1节至第6.4节的一些内容，若想

了解这些内容，则可复习这些章节。

## 14.1 分布的矩：均值、方差、偏斜度等

当一组数值有很强的集中趋势，即一种在某一特殊值周围聚集的趋势，则用与矩有关的一些量来表征这组数是非常有用的，**矩**则为这些量的整数幂之和。

最常见的是数值 $x_1,...,x_N$ 的均值：

$$\overline{x}=\frac{1}{N}\sum_{j=1}^{N}x_j \qquad (14.1.1)$$

$\overline{x}$ 是对所有在某一中心聚集值的估算，横杠表示平均，有时也用尖括号表示，即 $\langle x\rangle$。应当注意的是，均值并不是这种中心聚集值的唯一可用的估算量，也不一定是最佳的估算量。对于从拖有很宽"尾部"的概率分布中抽取的数值来说，其平均值的收敛性较差，或者当样本点数目增加时完全没有收敛性，这时，作为替代的估算量是**中位数**和**众数**，这两个量将在本节后面提到。

在表征了分布的中心值后，下面要讨论的常用量是分布围绕中心值的"宽度"和"变化量"，同样对此的测度不止一种量，最常用的是方差，

$$\mathrm{Var}(x_1...x_N)=\frac{1}{N-1}\sum_{j=1}^{N}(x_j-\overline{x})^2 \qquad (14.1.2)$$

或它的平方根，标准差

$$\sigma(x_1...x_N)=\sqrt{\mathrm{Var}(x_1...x_N)} \qquad (14.1.3)$$

式(14.1.2)估算了 $x$ 对其均值的均方差。关于式(14.1.2)中的分母为什么取 $N-1$而不取 $N$，有一段较长的故事，若读者不知这故事，可以参阅任何一本较好的统计学教材。如果对所测度的分布的变化的均值 $\overline{x}$ 是**先验**已知的，而不是从数据估算中得到的，则在式中采用 $N$ 代替 $N-1$。

由于均值依赖于数据的一阶矩，因此方差和标准差依赖于二阶矩。在实际生活中有时可以要处理二阶矩并不存在的分布(即为无穷大)，在这种情况下，方差和标准差作为对围绕中心值之数据宽度的度量，是无用的：随着样本点数的增加，从式(14.1.2)或(14.1.3)得到的值将不收敛，或从相同分布中抽取的数据组之间没有任何一致性。这种情况甚至出现在峰值宽度肉眼看起来相当有限的场合。一个更稳健的宽度估算量是**平均偏差**或**平均绝对偏差**，其定义为：

$$\mathrm{ADev}(x_1...x_N)=\frac{1}{N}\sum_{j=1}^{N}|x_j-\overline{x}| \qquad (14.1.4)$$

人们经常用样本中值 $x_{\mathrm{med}}$代替式(14.1.4)中的 $\overline{x}$，对任何固定的样本，中值实际上是使平均绝对偏差最小。

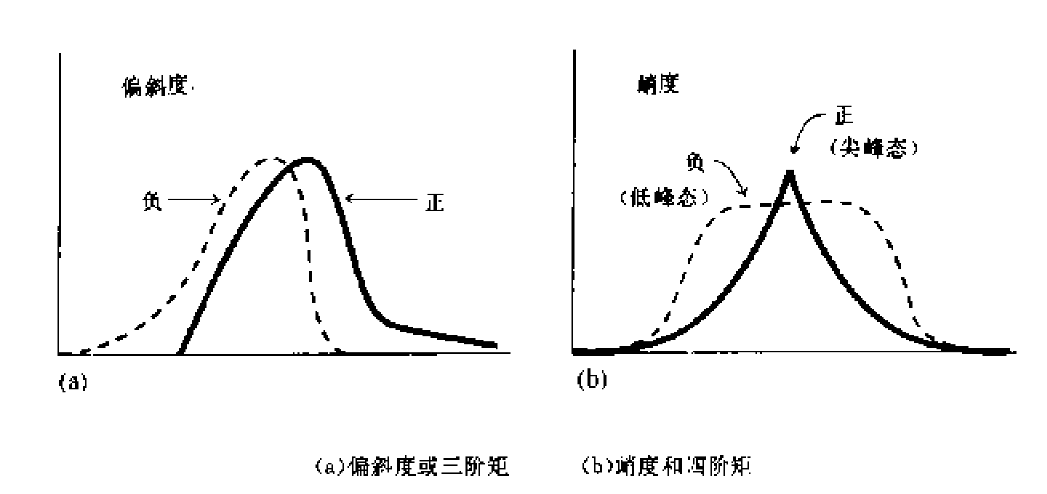
历史上统计学家早已对用式(14.1.4)代替式(14.1.2)一事嗤之以鼻，因为式(14.1.4)中的绝对值是"非解析的"，并且理论证明很困难。然而最近几年，传统已有改变，**稳健估算量**的问题已相当流行和重要(即估算较宽的分布，其具有相当显著的"分离"点)。较高的矩或涉及输入数据较高幂次的统计量，在稳健性上总是低于较低矩或仅涉及线性和或(所有最低矩)计算的统计量。

事实正是如此，**偏斜度即三阶矩**，和**峭度即四阶矩**应当谨慎使用，甚至最好不用。

偏斜度是对一分布围绕均值不对称程度的表征，尽管均值、标准差和平均偏差都是**有量纲**的量（即其与所测量 $x_j$ 有着相同的单位），而偏斜度的定义却使得其**无量纲**，其仅是一个纯粹的数来表征分布的形状，一般定义为：

$$\mathrm{Skew}(x_1,\ldots x_N)=\frac{1}{N}\sum_{j=1}^{N}\left[\frac{x_j-\bar{x}}{\sigma}\right]^3 \tag{14.1.5}$$

这里 $\sigma=\sigma(x_1,\ldots,x_N)$ 是分布的标准差（见式(14.1.3)），偏斜度的正值表示一个具有扩充到更多正 $x$ 的非对称尾部的分布；而负偏斜度表示尾部扩充到更多负 $x$ 的分布（见图14.1.1）。




(a)偏斜度或三阶矩　　(b)峭度和四阶矩

图14.1.1　三阶和四阶矩显著不同于正态分布的分布

当然，任何一组 $N$ 个测量值都可能给出非零的偏斜度值。即使其依从的分布是对称的（有零偏斜度）。为使式(14.1.5)有意义，我们必须对作为基本分布偏斜度的估算量之标准差有所了解。可惜的是，它仅取决于基本分布的形状，并且相当关键地取决于其尾端。对于理想的正态分布，式(14.1.5)的标准差近似为 $\sqrt{15/N}$。在实际中，仅当偏斜度是这一个数的很多倍时，相信偏斜度才有很好的实用性。

峭度也是一无量纲的量，用于测度分布的峰锐度或平坦度。相对于什么而言？当然是正态分布，除此还会有什么别的！一个带有正峭度的分布称之为**尖峰态**，麦特喇叭(Matterhorn)的轮廓线就是一个例子。一个带有负峭度的分布称之为**低峰态**，一个面包的轮廓即是它的一个例子（见图14.1.1），介于两者之间的称为**常峰态分布**。

峭度的常见定义为：

$$\mathrm{Kurt}(x_1,\ldots x_N)=\left\{\frac{1}{N}\sum_{j=1}^{N}\left[\frac{x_j-\bar{x}}{\sigma}\right]^4\right\}-3 \tag{14.1.6}$$

其中 −3 是为了使正态分布时式(14.1.6)为零值。

作为基本正态分布峭度的估算量(14.1.6)的标准差为 $\sqrt{96/N}$。然而，若峭度取决于那样一个有许多实际分布的高阶矩，那对这些分布来说，作为估计量的式(14.1.6)的标准差几乎为无穷大。

对这一章所定义的量的计算是相当直接的，许多教科书使用二项式定理，将定义式展开

成数据各种幂次的和，例如

$$\mathrm{Var}(x_1,\ldots,x_N)=\frac{1}{N-1}\left[\left(\sum_{j=1}^{N}x_j^2\right)-N\overline{x}^2\right]\approx\overline{x^2}-\overline{x}^2 \tag{14.1.7}$$

但这将增大由于较大因子造成的舍入误差，并且根据计算速度来看这也是不合理的。为了使舍入误差最小（特别是大的样本值），聪明的方式是使用校正的二步算法[2]，首先计算 $\overline{x}$，然后由下式计算 $\mathrm{Var}(x_1,\ldots x_N)$

$$\mathrm{Var}(x_1\ldots x_N)=\frac{1}{N-1}\left\{\sum_{j=1}^{N}(x_j-\overline{x})^2-\frac{1}{N}\left[\sum_{j=1}^{N}(x_j-\overline{x})\right]^2\right\} \tag{14.1.8}$$

如果 $\overline{x}$ 是实际值，第二项之和为零。除此之外，第一项将能很好地修正舍入误差。

```
#include <math.h>

void moment(float data[], int n, float *ave, float *adev, float *sdev,
    float *var, float *skew, float *curt)
    给定数组 data[1..n]，本程序返回均值 ave、平均偏差 adev、标准差 sdev、方差 var、偏斜度 skew 以及峭度 curt。
{
    void nrerror(char error_text[]);
    int j;
    float ep=0.0,s,p;

    if (n <= 1) nrerror("n must be at least 2 in moment");
    s=0.0;                                     第一步得到均值
    for (j=1;j<=n;j++) s += data[j];
    *ave=s/n;
    *adev=(*var)=(*skew)=(*curt)=0.0;          第二步得到偏离均值的一阶(绝对的)，二阶，
    for (j=1;j<=n;j++) {                           三阶和四阶矩
        *adev += fabs(s=data[j]-(*ave));
        *var += (p=s*s);
        *skew += (p *= s);
        *curt += (p *= s);
    }
    *adev /= n;
    *var=(*var-ep*ep/n)/(n-1);                 校正的二步公式
    *sdev=sqrt(*var);                          按照定义将这些部分结合在一起
    if (*var) {
        *skew /= (n*(*var)*(*sdev));
        *curt=(*curt)/(n*(*var)*(*var))-3.0;
    } else nrerror("No skew/kurtosis when variance = 0 (in moment)");
}
```

### 14.1.1 半不变量

独立随机变量之和的均值和方差是相加的：如果 $x$ 和 $y$ 分别取样于两个不同的概率分布，则

$$\overline{(x+y)}=\overline{x}+\overline{y}\quad \mathrm{Var}(x+y)=\mathrm{Var}(x)+\mathrm{Var}(x) \tag{14.1.9}$$

一般说来，高阶矩不是相加的，而它们一定的组合则是可以相加的，这种组合被称之为**半不变量**，如果一种分布的中心矩用 $M_k$ 表示：

$$M_k\equiv\langle(x_i-\overline{x})^k\rangle \tag{14.1.10}$$

因此，有 $M_2=\mathrm{var}(x)$，则第一类的几个半不变量 $I_k$ 由下式给出：

$$\begin{aligned}&I_2=M_2\qquad I_3=M_3\qquad I_4=M_4-3M_2^2\\&I_5=M_5-10M_2M_3\qquad I_6=M_6-15M_2M_4-10M_3^2+30M_2^3\end{aligned} \tag{14.1.11}$$

注意，偏斜度和峭度（式(14.1.5)和(14.1.6)）仅是半不变量的简单组合

$$\mathrm{Skew}(x)=I_3/I_2^{3/2}\qquad \mathrm{Kurt}(x)=I_4/I_2^2 \tag{14.1.12}$$

高斯分布所有高于$I_2$的半不变量都等于零，泊松分布所有半不变量等于其均值，更详尽的内容请参阅[2]。

### 14.1.2 中位数和众数

概率分布函数 $p(x)$ 的中位数为 $x_{med}$，对此值而言，较大的 $x$ 值和较小的 $x$ 值是等概率的：

$$\int_{-\infty}^{x_{med}} p(x)dx = \frac{1}{2} = \int_{x_{med}}^{\infty} p(x)dx \tag{14.1.13}$$

由样本值 $x_1,\ldots,x_N$ 来估价一种分布的中位数的方式是，找出居中的那个数 $x_i$，使其上下的样本数应相等。当然对于 $N$ 是偶数情形是不可能的，对这种情况往往用两个中心值的平均来作为中位数。如果 $x_j(j=1,\ldots,N)$ 以递增的形式排序，则中位数的公式为：

$$x_{med} = \begin{cases} x_{(N+1)/2} & N \text{ 为奇} \\ \frac{1}{2}(x_{N/2} + x_{N/2+1}) & N \text{ 为偶} \end{cases} \tag{14.1.14}$$

如果一种分布有较强的聚中趋势，使得它的大部分面积在某单一峰值下，则中位数是中心值的一种估算。它是比均值更为强的估算量：中位数仅不适用于尾端的面积较大的情况，而均值则不适用于尾端一阶矩较大的情况。很容易构造这样的例子，即使尾端的面积可忽略，其一阶矩还是较大的分布。

为了求得一组数据的中位数，人们首先要排序，然后利用式(14.1.14)。这是一个有 $N\log N$ 阶的过程。或许认为这是一种浪费，因为它产生了比中位数多得多的信息(即上下四分位点，十分位点等)。实际在第8.5节我们已看到由 $N$ 阶操作即可确定 $x_{(N+1)/2}$，可以参阅那一节的程序。

一个概率分布函数 $p(x)$ 的**众数**是 $p(x)$ 取极大值时的 $x$ 值。众数主要在概率分布有一个单一的尖锐最大值时很有用，在该情况下，众数估算了中心值。有时一种分布将是**双峰的**，即有两个极大值；那么人们希望知道两个单独的众数。注意在这种情况下，均值和中位数并不是非常用的，因为它们将仅能给出两峰值之间的"折衷"值。

参考文献和进一步读物：


Chan, T. F., Golub, G. H., and LeVeque, R. J. 1983, *American Staistician*, vol. 37, pp. 242～247. [1]

Cramer, H. 1946, *Mathematical Methods of Statistics* (Princeton: Princeton University Press), §15.10. [2]


## 14.2 两种分布具有相同的均值或方差吗？

我们很希望了解两种分布是否具有相同的均值。例如，第一组测量值是在某一事件前所收集的，而第二组测量值则在该事件以后收集的，那么这一事件(一种"处理"或"在控制参数上的变化")是否会造成两组值之间的差异。

第一个希望回答的问题是，一个样本的均值与另一个样本均值之间"有多大标准差"。实际上，这数值是相当有用的。如果均值之差是真实的，那么这一差异表述了这种均值之差的强度或"重要性"。然而，由其自身是无法说明这种差别是真实的(即统计上是否显著的)。如果数据点的数目较多，均值之差要远小于标准差，但仍然可能非常显著的；反之，如果数据是稀少的，这一差可能中等大小，但统计上并不显著。在以下几章，我们将会多次遇到强度和显

著性这些不同的概念。

测度均值之差显著性的量并不是区分它们的标准差之值，而是区分它们的**标准误之值**。一组数据的标准误测度了用样本均值来估算总体（或"真正"）均值的精度。一般说来，标准误等于样本的标准差除以该样本点数的平方根。

### 14.2.1 对于显著不同均值的学生 $t$ 检验

应用标准误的概念，测度均值之差显著性的常用统计量称之为**学生 $t$**。当两个分布被作为具有相同的方差、不同的均值时，学生 $t$ 按以下方式计算。首先，由以下公式从"合并方差"估算方差相同时，均值之差的标准误 $s_D$：

$$s_D = \sqrt{\frac{\sum_{i\in A}(x_i - \overline{x}_A)^2 + \sum_{i\in B}(x_i - \overline{x}_B)^2}{N_A + N_B - 2}\left(\frac{1}{N_A} + \frac{1}{N_B}\right)} \qquad (14.2.1)$$

其中求和是在一组样本的所有点上进行的，每组样本都各有自己的均值，$N_A$ 和 $N_B$ 分别为第一组和第二组样本的点数。其次，计算 $t$：

$$t = \frac{\overline{x}_A - \overline{x}_B}{s_D} \qquad (14.2.2)$$

第三步是利用方程(6.4.7)和(6.4.9)及第6.4节中的程序 **betai**（不完全 $B$ 函数）来计算 $t$ 值的显著性值，它服从自由度为 $N_A+N_B-2$ 的学生分布。

显著性用 0 到 1 之间的数来表征，并且是对于具有相等均值的分布其量 $|t|$ 恰好这样大或较大的概率。因此，小的显著性值（0.05 或 0.01）意味着所观察的差异是"非常明显的"。函数方程式(6.4.7)中的 $A(t/\nu)$ 等于 1 减去显著性。

```
#include <math.h>

void ttest(float data1[], unsigned long n1, float data2[], unsigned long n2,
    float *t, float *prob)
    给定数组 data1[1..n]和数组 data2[1..n2]，此程序将学生 t—检验返回为 t，将它的显著性返回到 prob，小的 prob
    值表明数组有显著不同的均值。数据数组假设取自于具有相同真实方差的总体。
{
    void avevar(float data[], unsigned long n, float *ave, float *var);
    float betai(float a, float b, float x);
    float var1, var2, svar, df, ave1, ave2;

    avevar(data1, n1, &ave1, &var1);
    avevar(data2, n2, &ave2, &var2);
    df=n1+n2-2;                                                    自由度
    svar=((n1-1)*var1+(n2-1)*var2)/df;                             合并方差
    *t=(ave1-ave2)/sqrt(svar*(1.0/n1+1.0/n2));
    *prob=betai(0.5*df,0.5,df/(df+(*t)*(*t)));                     参见式(6.4.9)
}
```

上述程序还使用了以下计算均值和方差的程序。

```
void avevar(float data[], unsigned long n, float *ave, float *var)
    给定数组 data[1..n]，返回它的均值为 ave 并且返回方差为 var。
{
    unsigned long j;
    float s,ep;
```

```
for (*ave=0.0,j=1;j<=n;j++) *ave += data[j];
*ave /=n;
*var=ep=0.0;
for (j=1;j<=n;j++) {
    s=data[j]-(*ave);
    ep += s;
    *var += s*s;
}
*var=(*var-ep*ep/n)/(n-1);           校正的二步公式(14.1.8)
}
```

以下考虑的情况是两种分布具有显著不同的方差，但我们仍然想知道它们的均值是否相同或不同。(一种医治脱发的治疗已经使某些患者头发全部脱落，但使其他患者成为毛人，则我们想知道，平均说来，这种疗法是否对治疗脱发有效!)对不相等方差的 $t$ 检验仍然有些疑问：如果两种分布有非常不同的方差，则它们在形状上可能极不相同，在这种情况下，均值的差异可能就不是特别有用的了。

为了找出两组数据集是否具有显著不同的方差，我们将使用本节后面叙述的 $F$ 检验。

对于不相等方差 t 检验的相关统计量为

$$t=\frac{\bar{x}_A-\bar{x}_B}{[\mathrm{Var}(x_A)/N_A+\mathrm{Var}(x_B)/N_B]^{1/2}} \tag{14.2.3}$$

这个统计量近似为学生 t 分布，自由度为

$$\frac{\left[\dfrac{\mathrm{Var}(x_A)}{N_A}+\dfrac{\mathrm{Var}(x_B)}{N_B}\right]^2}{\dfrac{[\mathrm{Var}(x_A)/N_A]^2}{N_A-1}+\dfrac{[\mathrm{Var}(x_B)/N_B]^2}{N_B-1}} \tag{14.2.4}$$

表达式(14.2.4)一般不是整数，但方程(6.4.7)并不介意这点。程序如下：

```
#include <math.h>
#include "nrutil.h"

void tutest(float data1[], unsigned long n1, float data2[], unsigned long n2,
    float *t, float *prob)
    给定数组 data1[1..n1]和 data2[1..n2]，这程序将学生 t 检验返回到变量 t，并且它的显著性返回到 prob，小的 prob
    值表明这些数组有显著不同的均值。数据数组允许取自于不同方差的总体。
{
    void avevar(float data[], unsigned long n, float *ave, float *var);
    float betai(float a, float b, float x);
    float var1, var2, df, ave1, ave2;
    avevar(data1, n1, &ave1, &var1);
    avevar(data2, n2, &ave2, &var2);
    *t=(ave1-ave2)/sqrt(var1/n1+var2/n2);
    df=SQR(var1/n1+var2/n2)/(SQR(var1/n1)/(n1-1)+SQR(var2/n2)/(n2-1));
    *prob=betai(0.5*df,0.5,df/(df+SQR(*t)));
}
```

我们最后一个例子是对配对样本情况的学生 $t$ 检验，我们设想在两个样本中，方差的许多部分是受到两个样本逐点对应的完全相同的效应，例如，有两个工作候选者，每个人都要有招聘委员会的 10 个相同的委员做出评价。我们希望知道十个得分数的均值是否显著不同。我们首先用上述程序 ttest 来试验，并得到一个并不特别显著的 prob 值(即>0.05)，但或许因为某些委员总是给高分，而其它委员总是给低分，这种倾向使显著性被消除。它增加

了明显的方差，并减少了均值上任何不同的显著性。因此我们有：

$$\mathrm{Cov}(x_A,x_B) = \frac{1}{N-1}\sum_{i=1}^{N}(x_{Ai}-\bar{x}_A)(x_{Bi}-\bar{x}_B) \tag{14.2.5}$$

$$s_D = \left[\frac{\mathrm{Var}(x_A)+\mathrm{Var}(x_B)-2\mathrm{Cov}(x_A,x_B)}{N}\right]^{1/2} \tag{14.2.6}$$

$$t = \frac{\bar{x}_A-\bar{x}_B}{s_D} \tag{14.2.7}$$

其中 $N$ 是每组样本的点数(配对数)。注意，重要的是每个特别的 $i$ 值表示每个样本中的相应点，即配对的点数。在式(14.2.7)中，$t$ 统计量的显著性被计算，仅有 $N-1$ 个自由度。程序如下：

```
#include <math.h>

void tptest(float data1[], float data2[], unsigned long n2,
    float *t, float *prob)
    给定配对数组 data1[1..n]和 data2[1..n]，本程序将配对数据的学生 t 一检验返回到 t，将它的显著性返回到 prob，
    小值的 prob 表示均值有显著差别。
{
    void avevar(float data[], unsigned long n, float *ave, float *var);
    float betai(float a, float b, float x);
    unsingd long j;
    float var1, var2, ave1, ave2, sd, df, cov=0.0;

    avevar(data1, n, &ave1, &var1);
    avevar(data2, n, &ave2, &var2);
    for (j=1;j<=n; j++)
        cov += (data1[j]-ave1) * (data2[j]-ave2);
    cov /= df=n-1;
    sd=sqrt((var1+var2=2.0 * cov)/n)
    *t=(ave1-ave2)/sd;
    *prob=betai (0.5 * df,0.5,df/(df+(*t) * (*t)));
}
```

## 14.2.2 对显著不同方差 $F$ 检验

$F$一检验是验证两个样本具有不同的方差的假设，它是通过否定它们方差实际上是一致的这一无效假设来进行的。统计量 $F$ 是一个方差对另一个方差的比值，其值(或 >>1 或 <<1)将指出非常显著的差异。它是利用 Betai 程序估值的。在一般情况下，我们可以通过或者非常大的 $F$ 值或非常小的 $F$ 值来证明无效假设(相同方差)不成立，因此正确的显著性是双尾的，即两个不完整 Betai 函数的和。通过式(6.4.3)可以证明两个尾端是相等的；因此我们仅需要计算一个，然后加倍它。偶而，当"无效假设"特别可行时，双尾的一致性变得模糊，给出了一个大于1的指示概率。这时改变概率为2减去其自身，以使正确地交换尾端。这些考虑和方程(6.4.3)给出了如下程序：

```
void ftest(float data1[], unsigned n1, float data2[], unsigned long n2,
    float *f, float *prob)
    给定数组 data1[1..n1]和 data2[1..n2]，程序返回 f 值和它的显著性 prob。小的 prob 值表示这两个数组有显著不
    同的方差。
{
    void avevar(float data[], unsigned long n, float *ave, float *var);
```

```
    float betai(float a, float b, float x);
    float var1, var2, ave1, ave2, df1, df2;

    avevar(data1, &ave1, &var1);
    avevar(data2, &ave2, &var2);
    if (var1 > var2) {                    使F成为较大方差与较小方差之比
        *f=var1/var2;
        df1=n1-1;
        df2=n2-1;
    }else {
        *f=var2/var1;
        df1=n2-1;
        df2=n1-1;
    }
    *prob = 2.0*betai(0.5*df2,0.5*df1,df2/(df2+df1*(*f)));
    if (*prob > 1.0) *prob=2.0-*prob;
}
```

## 14.3 两种分布是否不同?

给出两组数据,我们能把上节提到的问题更一般化并仅提出一个问题:两组数据是否是从同一分布函数中抽取的,或者从不同分布函数中抽取的?用适当的统计学语言来讲,即:"在一定要求的显著性下我们是否能推翻'无效假设',即两组数据集是从同一总体分布函数中抽取的假说。"否定了无效假设实际上就证明了数据集来自不同的分布,另外如果不能否定无效假设,也仅仅能证明数据集与一个单一的分布函数是一致的(**相容的**),人们永远不能证明两组数据来自于单个的分布,因为,例如没有实际的数据量能够区分两种分布,这两种分布仅相差 $10^{12}$分之一的量级。

证明两种分布互不相同的,或者表明它们是相容的是件一直出现在许多研究领域的任务:可见的星体是否均匀分布在天空中(即星体在天体中的位置是作为赤纬的函数分布与天体区域作为赤纬函数的分布是相同的吗?)布鲁科林(Brooklyn)的教育模式是否与布鲁尼克斯(Bronx)的相同呢?(即人的分布作为最后阶段受教育的函数是否相同?)两种牌号的荧光灯使用寿命的分布是否相同?对第一胎、第二胎、第三胎等等的孩子,他们的水痘的发生率是否相同?

这四个例子表明了以下两种不同的两分法的四种组合,这两种两分性为:(1)数据是连续的或离散的。(2)我们要么希望比较一组数据与一已知分布,要么希望比较两个同样的未知分布的数据集。关于荧光灯的寿命和星体的数据集是连续的,因为我们能给出各个荧光灯的寿命和星球位置的表格。而水痘和教育水平则离散的,因为我们得到的是以离散形式给出表:第一胎、第二胎等,或者6年级、7年级等。另外,星球和水痘的"无效假设"是一个已知分布(天空中区域分布和在总人口中水痘的发生率)。而荧光灯和教育水准则涉及到两个同样都是未知数据组的分布的比较(两种牌号,或 Brooklyn 和 bronx)。

人们总是能够将连续数据转化为量化的数据,只要通过以下方式,即将事件分成不同的组,每一组的连续量有特定的范围:赤纬度是 1~10 度、10~20 度、20~30 度等。离散量化过程总是损失一定的信息,不管怎样,量化级的选择也有相当的任意性。与其他研究者一样,我们也希望避免不必要的离散化。

对于离散分布之间的差异,可以接受的检验是 $\chi^2$**检验**。对于单变量函数的连续数据组,

最常用的检验是柯尔莫哥洛夫—史密诺夫(Kolmogorov—Smirnov)检验。我们将依次予以讨论。

### 14.3.1 X平方检验

假设观察到第$i$个量化区域内的数目为$N_i$，并且根据某一已知分布所预期的数应为$n_i$，注意$N_i$是整数，$n_i$则未必是整数，则$\chi^2$统计量为：

$$\chi^2 = \sum_i \frac{(N_i - n_i)^2}{n_i} \tag{14.3.1}$$

其中求和是对所有量化级的，较大的$\chi^2$值指出“无效假设”(即$N_i$是从以$n_i$表征的母体分布中抽取的)是相当不可能的。

式(14.3.1)中具有$0=n_j=N_j$的任何第$j$项都应从和式中略去，如$n_j=0$，$N_j\neq 0$，则$\chi^2$趋于无穷大，这是对的，因为在这种情况下$N_j$所代表的离散值不可能从$n_j$所表征的总体中抽取。

$\chi^2$**概率函数**$Q(\chi^2|\nu)$是一个不完全的伽马函数，这已在第6.2节中讨论过(见式(6.2.18))。严格地说，$Q(\chi^2|\nu)$是$\nu$个具有单位方差的随机正态变量(均值为零)的平方和大于$\chi^2$的概率。在式(14.3.1)求和中的项并不是个个为正态分布。然而，如果量化级的数较大(>>1)或每一量化层中事件出现的次数较大时(>>1)，则在无效假设情况下$\chi^2$分布是对式(14.3.1)的一个很好的近似，用它来估计$\chi$平方检验的显著性是标准的。

适当的$\nu$值(自由度数)，提供了进一步的讨论。如果以$n_i$表征的固定模型来收集数据，即不再为拟合总的观察数$\sum N_i$而重新归一化，则$\nu$等于离散量的总数$N_B$(注意$N_B$**并不是**事件的总数)。如果$n_i$事后被归一化，以致它的和等于$N_i$的和，则$\nu$的值等于$N_B-1$，并且称模型有一约束(在下面程序中 Knstrn=1)。如果模型有一附加的自由参数，则这些附加的“拟合”参数将减小$\nu$(或增加约束)一个附加单位。于是我们有如下程序。

```
ovid chsone(float bins[], float ebins, int knstrn, float *df,
    float *chsq, float *prob)
    给定数组bins[1..nbins]包含事件的观测数，并给定数组ebins[1..nbins]，包含事件的所期望的数，以及给定的约束个数knstrn(通常为1)，本程序(平凡地)返回自由度的个数df，并且(非平凡地)将χ²返回到chsq和显著性返回到prob，小的prob值表示分布bins和ebins之间有显著的差别，注意bins和ebins都是浮点型数组，虽然通常bins是整数值。
{
    float gammq(float a, float x);
    void nrerror(char error_text[]);
    int j;
    float temp;

    *df=nbins-knstrn;
    *chsq=0.0;
    for (j=1;j<=nbins;j++) {
        if (ebins[j] <= 0.0) nrerror("Bad expected number in chsone");
        temp=bins[j]-ebins[j];
        *chsq += temp*temp/ebins[j];
    }
    *prob=gammq(0.5*(*df),0.5*(*chsq));          χ²概率函数，见第6.2节
}
```

下面我们考虑比较两个离散数据集，设 $R_i$ 为第一组数据集第 $i$ 个量化级中出现的数目，$S_i$ 为第二组数据集同样第 $i$ 个量化级中出现的数目，则 $\chi^2$ 统计量为：

$$\chi^2 = \sum_i \frac{(R_i - S_i)^2}{R_i + S_i} \tag{14.3.2}$$

比较式(14.3.2)和(14.3.1)，应当注意式中分母并不是 $R_i$ 和 $S_i$ 的平均(它是式(14.3.1)中 $n_i$ 的一个估计量)。更确切地说，它两倍于平均(和)，原因是在 $\chi^2$ 和式中的每一项都假设近似方差为1的正态分布量的平方。两正态量之差的方差等于它们各自方差的和，而不是平均。

如果数据是以这样的方式收集的，即 $R_i$ 和式等于 $S_i$ 的和式，则自由度的个数等于量化级的数目减一，$N_B-1$(即 knstrn$-1$)。如果没有这一要求，自由度个数则为 $N_B$。例如，一位鸟类观察家想了解今年观察到的鸟种类的分布是否与去年相同。每一物种相当于量化级。如果观察者仅取他每年中首先看到的 1000 只鸟，则自由度的个数为 $N_B-1$。应注意后一种情况中，观察者也还在检验这一年的鸟是否比另一年的多，这是一额外的自由度。当然对数据组任何附加的约束都会降低自由度(即增加 Knstrn 到一个更正的值)。程序为：

```
void chstwo(float bins1[], float bins2[], int nbins, int knstrn,
    float *df, float *chsq, float *prob)
    给定数组 bins1[1..nbins]和数组 bins2[1..nbins]，它包含两组离散数据集，并且已知附加约束数 knstrn(正常情况为 0 或 1)，本程序返回自由度的个数 df，χ²检测的 chsq 和显著性 prob。小值的 prob 表示 bins1 和 bins2 两分布之间有显著差别。注意 bins1 和 bins2 是浮点型数组，尽管它们通常是整数值。
{
    float gammq(float a, float x);
    int j;
    float temp;

    *df=nbins-knstrn;
    *chsq=0.0;
    for (j=1;j<=nbins;j++)
        if (bins1[j] == 0.0 && bins2[j] == 0.0)
            --(*df);                                没有数据意味着自由度少1
        else {
            temp=bins1[j]-bins2[j];
            *chsq += temp*temp/(bins1[j]+bins2[j]);
        }
    *prob=gammq(0.5*(*df),0.5*(*chsq));        χ²概率函数，见第6.2节
}
```

方程(14.3.2)和程序 chstwo 都用于如下情况，即两离散数据集有相同的总数据数。对于不相同的情况，则有：

$$\chi^2 = \sum_i \frac{(\sqrt{S/R}R_i - \sqrt{R/S}S_i)^2}{R_i + S_i} \tag{14.3.3}$$

其中

$$R \equiv \sum_i R_i \qquad S \equiv \sum_i S_i \tag{14.3.4}$$

分别是数据点的对应数。其程序可将 chstwo 适当改变而得到。

### 14.3.2 K-S 检验 (Kolmogorov-Smirnov Test)

K-S 检验主要用于非离散量化分布，其为某一独立变量的函数，即对数据集来说，每一数据点都与一个单一的数相联系(如每一灯泡的寿命或每一星球的赤纬度)。在这种情况下，

数据集很容易被变换成一被抽取概率分布的累积分布函数的无偏估计量 $S_N(x)$：如果 $N$ 个事件分别位于 $x_i(i=1,\ldots,N)$，则 $S_N(x)$ 是一函数，其给出各在每一给定 $x$ 点的左边数据点的分数，这一函数在两连续点 $x_i$ 之间是常数（即分成递增顺序），并且在每一点 $x_i$ 跳动 $1/N$ 常数（见图14.3.1）。

不同的分布，或数据集，由上面过程，给出不同的累积分布函数估计。然而所有累积分布函数在 $x$ 的最小允许值（其为零）处和最大允许值（其为 1）处是一致的（当然最大值和最小值为 $\pm\infty$）。因这正是最小值和最大值之间的行为来区别分布。

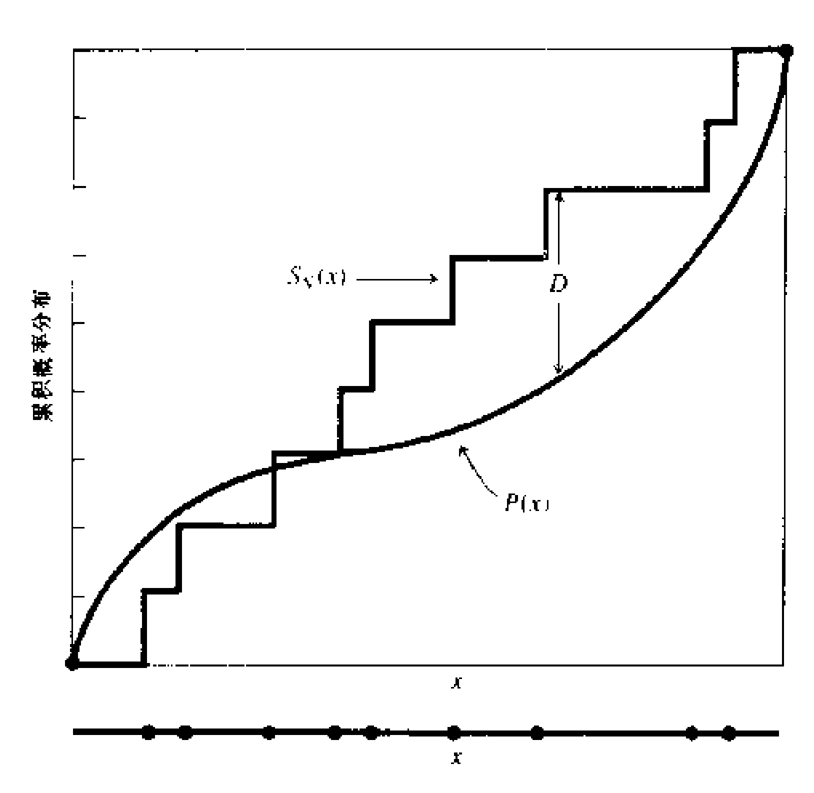
人们能够想到用任何统计量来测度两累积分布之间整体的差异：例如两者之间面积的绝对值，或者它们积分均方差。K-S 检验中 $D$ 是一种特别简单的测度：它被定义为两累积分布函数绝对差的最大值。因此，对比较一组数据集的 $S_N(x)$ 与一已知分布累积函数 $P(x)$ 来说，K-S 统计量是：

$$D = \max_{-\infty<x<\infty} |S_N(x) - P(x)| \tag{14.3.5}$$

而比较两个不同的累积分布函数 $S_{N_1}(x)$ 和 $S_{N_2}(x)$，则 K-S 统计量为

$$D = \max_{-\infty<x<\infty} |S_{N_1}(x) - S_{N_2}(x)| \tag{14.3.6}$$

使得 K－S 统计量有用的是在"无效假设"情况下（即数据组抽样于同一分布），它的分布能够计算，至少是有用的近似，因此给出了任何非零 $D$ 的观察值的显著性。K－S 检验的主要特点是在 $x$ 重新参数化过程中的不变性，换言之，人们能够局部地滑动和伸缩图14.3.1中的 $x$ 轴，而最大距离 $D$ 保持不变。例如，使用 $\log x$ 与使用 $x$ 可以得到相同的显著性。




在 $x$ 点的已测分布值（如图下面的横坐标上 $N$ 个点）与画成 $P(x)$ 的理论累积概率分布值进行比较。构造了阶梯型累积分布 $S_N(x)$，这是一个在每个测量点等量上升的分布。$D$ 是这两个累积分布之间的最大距离。

图14.3.1　K-S统计量 $D$

参与显著性计算的函数可写成如下和式：

$$Q_{KS}(\lambda) = 2\sum_{j=1}^{\infty}(-)^{j-1}e^{-2j^2\lambda^2} \tag{14.3.7}$$

这是一个单调函数，极限值为

$$Q_{KS}(0) = 1 \qquad Q_{KS}(\infty) = 0 \tag{14.3.8}$$

根据这个函数，$D$ 观察值的显著性水平近似由下式给出：

$$\text{概率}\ (D > \text{观察值}) = Q_{KS}\left(\left[\sqrt{N_e} + 0.12 + 0.11/\sqrt{N_e}\right]D\right) \tag{14.3.9}$$

其中，$N_e$ 是数据点的有效个数，对式(14.3.5)仅有一种分布时，$N_e=N$，而对式(14.3.6)有两种分布情况时：

$$N_e = \frac{N_1N_2}{N_1+N_2} \tag{14.3.10}$$

其中，$N_1$是第一个分布中数据点的个数，$N_2$是第二个分布中数据点的个数。

包含在式(14.3.9)中近似的性质是当 $N_e$ 变大时，它也渐近精确。但对 $N_e \geqslant 4$ 时，它已足够好，就如同人们曾可能使用的同样的小的数(参见[1])。

因此，我们有以下程序：

```
#include <math.h>
#include "nrutil.h"

ovid ksone(float data[], unsigned long n, float (*func)(float), float *d,
    float *prob)
    给定数组 data[1..n]，并给定一个用户提供的单变量的函数 func，它是一个范围从0(其变量的最小值)到1(其变量
    的最大值)的累积分布概率，本程序返回 K-S 统计量 d 以及显著性水平 prob。小的 prob 值表明 data 的累积分布函
    数明显不同于 func。数组 data 被按递增次序排列而修正。
{
    float probks(float alam);
    void sort(unsigned long n, float arr[]);
    unsigned long j;
    float dt,en,ff,fn,fo=0.0;

    sort(n,data);                                    如果数据已经按递增排序，则此调用可以省略
    en=n;
    *d=0.0;
    for (j=1;j<=n;j++) {                             关于排序数据点的循环
        fn=j/en;                                     在这步以后，数据的累积分布函数
        ff=(*func)(data[j]);                         与用户提供的函数比较
        dt=FMAX(fabs(fo-ff),fabs(fn-ff));            最大距离
        if (dt > *d) *d=dt;
        fo=fn;
    }
    en=sqrt(en);
    *prob=probks((en+0.12+0.11/en)*(*d));            计算显著性
}

    #include <math.h>

void kstwo(float data1[], unsigned long n1, float data2[], unsinged long n2,
    float *d, float *prob)
    给定数组 data1[1..n1]和数组 data[1..n2]，本程序返回 K-S 统计量 d，以及关于数据集取自于相同分布的无效假
    设的显著性水平 prob。小的 prob 表明 data1的累积分布函数显著不同于 data2的累积分布函数，数组 data1 和 data2
    修改为按递增顺序排序。
```

```
    float probks(float alam);
    void sort(unsigned long n, float arr[]);
    unsigned long j1=1,j2=1;
    float d1,d2,dt,en1,en2,en,fn1=0.0,fn2=0.0;
    sort(n1,data1);
    sort(n2,data2);
    en1=n1;
    en2=2;
    *d=0.0;
    while (j1 <= n1 && j2 <= n2) {                              如果我们没做完...
        if ((d1=data1[j1]) <= (d2=data2[j2])) fn1=j1++/en1;     下一步在data1中
        if (d2 <= d1) fn2=j2++/en2;                             下一步在data2中
        if ((dt=fabs(fn2-fn1)) > *d) *d=dt;
    }
    en=sqrt(en1*en2/(en1+en2));
    *prob=probks((en+0.12+0.11/en)*(*d));                       计算显著性
}
```

上述两个程序都使用了如下程序进行函数 $Q_{KS}$ 的计算：

```
#include <math.h>
#define EPS1 0.001
#define EPS2 1.0e-8

float probks(float alam)          K-S概率函数
{
    int j;
    float a2,fac=2.0,sum=0.0,term,termbf=0.0;

    a2 = -2.0*alam*alam;
    for (j=1;j<=100;j++) {
        term=fac*exp(a2*j*j);
        if (fabs(term) <= EPS1*termbf || fabs(term) <= EPS2*sum) return sum;
        fac = -fac;                       变换和式中的符号
        termbf=fabs(term);
    }
    return 1.0;                           仅仅因为不收敛而到达此
}
```

## 14.3.3 K-S检验的变形

K-S检验对一个积累分布函数 $P(x)$ 偏差的灵敏度是不独立于 x 的。实际上，K-S检验在中位数附近最灵敏，即 $P(x)=0.5$；在分布的末端最不灵敏，$P(x)$ 接近 0 或 1。原因是在无效假设中，$|S_N(x)-P(x)|$ 并没有独立于变量 $x$ 的概率分布。更正确地说，它的变化正比于 $P(x)[1-P(x)]$，其在 $P=0.5$ 处取最大值。因为式(14.3.5)中的K-S统计量是两累积分布函数在整个 $x$ 轴的最大差值，一个在其自身 $x$ 值上统计显著的偏差，将变得可与 $P=0.5$ 处所希望的随机偏差相比拟，因此它是打折扣的。结果是K-S检验善于寻找一种概率分布的偏移，特别是中位数的改变，但它并不总是善于寻找发散，其更影响概率分布的末端，而使中位数保持不变。

增加K-S统计量在末端作用的一种方式是，用一个称之为稳定的或加权的统计量[2-4]代替式(14.3.5)中的 $D$，例如 Anderson-Darling 统计量

$$D^* = \max_{-\infty<x<\infty} \frac{|S_N(x)-P(x)|}{\sqrt{P(x)[1-P(x)]}} \tag{14.3.11}$$

遗憾的是，对式(14.3.7)和(14.3.9)没有相应的简单公式，虽然 Noé[5] 利用递推关系给出了一种计算方法，

并提供了数值结果[2]。还有许多其它可能相似的统计量，例如

$$D^{**}=\int_{P=0}^{1}\frac{|S_N(x)-P(x)|}{\sqrt{P(x)[1-P(x)]}}dP(x) \tag{14.3.12}$$

也是由 Anderson 和 Darling[3] 加以讨论的。

另一种更简单更直接的方法应归于 Kuiper[6,7]。我们已经知道标准 K-S 检验在变量 $x$ 的重新参数化下是不变的。一个更一般的对称性(其保证在所有 $x$ 值上有相等的灵敏度)是环绕 $x$ 轴成一个圆($\pm\infty$ 点是等价的)，并且寻找一个统计量，其在该圆上所有偏移和参数化下都是不变的。例如，它允许在 $x$ 的某一中心值"切断"概率分布并交换左右两边的 $x$ 值，统计量和其显著性都没有变化。

Kuiper 的统计量定义为：

$$V=D_{+}+D_{-}=\max_{-\infty<x<\infty}[S_N(x)-P(x)]+\max_{-\infty<x<\infty}[P(x)-S_N(x)] \tag{14.3.13}$$

它是 $S_N(x)$ 在 $P(x)$ 上面和下面最大距离之和，应相信这一统计量在圆上具有所希望的不变性：因为角度可超过 360° 许多次，因此可以作为围绕所张角度的函数，而画出定义在圆上两概率分布的无穷限积分，如果改变积分的起点，$D_+$ 和 $D_-$ 将发生变化，但它们的和则保持不变

因此，对于统计量 $V$ 的渐近分布有一简单的公式，可直接模拟(14.3.7)～(14.3.10)，设

$$Q_{KP}(\lambda)=2\sum_{j=1}^{\infty}(4j^2\lambda^2-1)e^{-2j^2\lambda^2} \tag{14.3.14}$$

它是单调的，并满足：

$$Q_{KP}(0)=1\qquad Q_{KP}(\infty)=0 \tag{14.3.15}$$

根据这一函数，显著性水平为[1]：

$$\text{概率}\ (V>\text{观察值})=Q_{KP}([\sqrt{N_e}+0.155+0.24/\sqrt{N_e}]D) \tag{14.3.16}$$

这里，在一组样本情况下，$N_e$ 等于 $N$；而在两组样本情况下，则为式(14.3.10)的值。

当然，Kuiper 检验对在一个圆上定义的任何问题都是理想的，例如检验某些东西的经度是否与理论一致，或两个东西在经度是否有不同的分布(见[8])。

我们将类似于 ksone、kstwo 和 probks 程序的编码留给读者去做。(对 $\lambda<0.4$，不要试图对式(14.3.14)求和，它的值是 1 至 7 的数字，但该级数需要许多项才能收敛，并失去舍入精度。)

两个最后应当引起注意的问题是：第一是所有 K-S 检验都缺少区分某种分布的能力，一个简单的例子是，一带有窄"凹口"的分布，在"凹口"内概率为零，当然在这一"凹口"内甚至有一数据点的存在都会将这一分布排除，但是因为它的积累性质，K-S 检验在对偏差预告之前需要在"凹口"内有许多数据点。

其次，我们应注意，如果我们从一数据集中估算任何参数(即均值和方差)，则对于使用了这种估计参数的积累分布函数 $P(x)$，K-S 统计量 $D$ 的分布再也不是由式(14.3.9)给出。一般来说，人们必须自己来决定新的分布，如利用蒙特卡罗法(Monte Carlo)。

## 参考文献和进一步读物：


Stephens, M.A. 1970, *Journal of the Royal Statistical Society*, ser. B, vol. 32, pp. 115～122. [1]

Anderson, T.W., and Darling, D.A. 1952, *Annals of Mathematical Statistics*, vol. 23, pp. 193～212. [2]

Darling, D.A. 1957, *Annals of Mathematical Statistics*, vol. 28, pp. 823～838. [3]

Michael, J.R. 1983, *Biometrika*, vol. 70, no. 1, pp. 11～17. [4]

Noe, M. 1972, *Annals of Mathematical Statistics*, vol. 43, pp. 58～64. [5]

Kuiper, N.H. 1962, *Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen*, ser. A., vol. 63, pp. 38～47. [6]

Stephens, M.A. 1965, *Biometrika*, vol. 52, pp. 309～321. [7]

Fisher, N. I., Lewis, T., and Embleton, B. J. J. 1987, *Statistical Analysis of Spherical Data* (New York: Cambridge University Press). [8]

## 14.4 两种分布的列联表分析

在这一节和下两节，我们将处理两种分布之间**关联的测度**。情况是这样的：每个数据点都有两个或更多个与它相联系的不同的量，并且我们也希望知道是否对某一量的了解有利于我们预报另一量的值。在许多情况下，一个变量是"独立"或"控制"变量，而另一变量则是"非独立"或"观测"变量。那么，我们希望知道是否后一变量实际上依赖于或关联于前一变量。如果确实如此，我们希望对这一关联的强度有定量的测度。人们通常不太严谨地将其表述为两变量是**相关**还是**不相关**，但我们将保留那些特殊关联（线性、或者至少是单调的）的术语，这将在第14.5节和14.6节予以讨论。

应当注意的是，就象以前几节一样，显著性和强度的不同概念表现在：即使两种分布之间的关联相当弱，如果数据量足够大，这种关联仍是非常显著的。

用不同类型形成松散的层次结构来区分某些不同种类的变量是很有用的。

- 一个变量，当它的值是某一无序集合的成员时，则称其为**公称的**。例如："居住的州"是一个公称变量，在美国，这是取50个值中的一个；在天文学上，"星系的类型"是一个公称变量，它取三个值："螺旋星系"、"椭圆星系"和"不规则星系"
- 当一变量的值是某一离散、而有序集合的元素时，则称该变量为**有序的**。例如，学校的年级、太阳系行星的次序（水星＝1，金星＝2，…）、子孙的数目等，在一有序变量的值之间不必有"相等度量距离"的概念，而仅需要它们是内在有序的。
- 我们将称一变量为**连续的**，如果它的值是实数，如距离、时间、温度等（有时社会学家还区别区间和比率连续变量，但我们并不认为这种区分是必需的）。

一个连续变量可以量化为有序量，如果忽略其顺序，则有序量又可变为公称变量。公称变量构成了结构为最低层次，因而也是最一般的。例如，一组几个连续或有序变量的集合可以粗糙地转变成单个的公称变量，即先量化分级，后将每个离散的量化级看成一个单一的公称变量，当多维数据较稀疏时，这是唯一明智的处理方式。

本节余下部分将讨论两个公称变量之间关联的测度。对任何一对公称变量，数据能用一**列联表**来显示，该表的行由一个公称变量的值来标号，列由另一个公称变量值来标号，表的项目是非负整数，它们给出了行和列的每一组合的观测事件的数目（见图14.4.1）。这种对公称变量之间关联的分析称为列联表分析或交叉列表分析。

我们将引入两种不同的方法，第一种方法是基于$\chi$平方统计量，它能较好地表征这种关联的显著性，但它用作为强度的测度却很一般（主要由于其数值没有任何非常直接的解释）。第二种方法是基于信息论中熵的概念，它根本没有引入关联显著性（仅对$\chi$平方而言），但它能非常明确地表征已知很显著的关联强度。

### 14.4.1 基于$\chi^2$的关联测度

首先，我们设$N_{ij}$表示第一个变量取第$i$个值、第二个变量取第$j$个值时事件发生的数目。设$N$表示总事件数，即等于所有$N_{ij}$的和。设$N_{i\cdot}$表示第一变量$x$取第$i$值而不管第二变

|  | 1. 红 | 2. 绿 | ... |  |
|---|---|---|---|---|
| 1. 男 | 红女 $N_{11}$ | 绿男 $N_{12}$ | ... | 男性 $N_{1\cdot}$ |
| 2. 女 | 红男 $N_{21}$ | 绿女 $N_{22}$ | ... | 女性 $N_{2\cdot}$ |
| ⋮ | ... | ... | ... | ⋮ |
|  | 红 $N_{\cdot 1}$ | 绿 $N_{\cdot 2}$ | ... | 总数 $N$ |

行和列的旁注(总数)也被给出.变量是"公称"的,即它们的值排列的次序是任意的,并且不会影响列表结果。如果值次序有一定内在的意义,则变量是"有序的"或"连续的",并且第14.5节～第14.6节的相关分析可以使用。

图14.4.1　两个公称变量(这里是性别和颜色)列联表的例证

量 $y$ 取何值时事件发生的数目，$N_{\cdot j}$ 表示 $y$ 取第 $j$ 个值而不管 $x$ 取何值时事件发生的数目，因此我们有：

$$N_{i\cdot} = \sum_j N_{ij} \qquad N_{\cdot j} = \sum_i N_{ij}$$
$$N = \sum_i N_{i\cdot} = \sum_j N_{\cdot j} \tag{14.4.1}$$

$N_{\cdot j}$ 和 $N_{i\cdot}$ 有时称之为**行和列总数**或**边缘值**，我们将谨慎使用这些术语。因为我们不能直接确定哪个是行，哪个是列。

无效假设是两变量 $x$ 和 $y$ 没有联系。在这种情况下，给定某一 $y$ 值时某一 $x$ 值的概率，等于不管 $y$ 取何值时 $x$ 取该值的概率。因此，在无效假设的情况下，对任何 $N_{ij}$ 的期望的数(记为 $n_{ij}$)可仅从行和列总数中计算出来：

$$\frac{n_{ij}}{N_{\cdot j}} = \frac{N_{i\cdot}}{N} \quad \text{意味着} \quad n_{ij} = \frac{N_{i\cdot} N_{\cdot j}}{N} \tag{14.4.2}$$

注意，如果某一列或行的总数为零，则在那一列或行中所有项的期望数都为零，在这种情况下，从不发生的 $x$ 或 $y$ 的量化级将从分析中简单地除去。

$\chi^2$ 统计量现在由式(14.3.1)给出，在目前情况下，它是表中所有项的和

$$\chi^2 = \sum_{i,j} \frac{(N_{ij} - n_{ij})^2}{n_{ij}} \tag{14.4.3}$$

自由度的个数等于表中项目的数目(即行和列大小的乘积)减去约束条件数(即我们用它来决定 $n_{ij}$ 的)，每个行总数和列总数都是约束条件(除去已多计算了一个)。因为行和列的总数都等于 $N$，即数据点的总数。因此，如果表的大小是 $I \times J$，则自由度数等于 $IJ - I - J + 1$。式(14.4.3)连同 $\chi^2$ 概率函数(第6.2节)给出了变量 $x$ 和 $y$ 之间关联的显著性。

假如有显著的联系，那么我们如何才能定量刻划它的强度，即我们是否能比较一个关联与另一关联的强度?这里的想法是找到某一重新参数化的 $\chi^2$，其映射自身到更适当的区间，从 0 到 1，其中的结果并不取决于我们取样数据的量，而取决于数据取样的总体。有几种不同的方式可做到这一点。最常见的两种是**克莱姆(Cramer)的 V 和列联系数 C**。

Cramer 的 $V$ 定义为：

$$V = \sqrt{\frac{\chi^2}{N\min(I-1, J-1)}} \tag{14.4.4}$$

其中，$I$ 和 $J$ 是行和列的数目，$N$ 是事件的总数。Cramer V 具有位于 0 和 1 之间(包括 0 和 1)令人愉快的性质。当没有关联时等于 0，关联完备时等于1：任意行中所有事件都位于唯一的列中，反之亦然。(按(国际)象棋术语说，任何两个置于非零表项上的车，都不能互相拼杀)。在 $I=J=2$ 的情况下，$V$ 称之为 Phi 统计量。

列联系数 $C$ 定义为：

$$C = \sqrt{\frac{\chi^2}{\chi^2 + N}} \tag{14.4.5}$$

它也位于 0 和 1 之间，但它不能达到它的上限，当它被用于比较带有相同 $I$ 和 $J$ 的两个表的强度时，它的上限取决于 $I$ 和 $J$，因此，它不能用于比较两不同的大小的表。

$V$ 和 $C$ 的麻烦是：当它们取两端部之间的值时，不能直接解释这些值意味着什么。例如，你在拉斯维加斯，一位朋友告诉你，在赌场管理员的眼睛与它们的赌盘红和黑出现的机会之间有一虽小但显著的关系，并告诉你 $V$ 值是 0.028。那么，你知道什么样的运气在等着你，这种关联是否能使你赢钱?请不要来问我们，不知道!

```
#include <math.h>
#include "nrutil.h"
#define TINY 1.0e-30

void cntab1(int **nn, int ni, int nj, float *chisq, float *df,
    float *prob,float *cramrv, float *ccc)
    给定呈整数型式 nn[1..ni][1..nj]的二维列联表，程序返回 χ²的 chisq，自由度个数 df，显著性水平 prob(小的值表
    示显著关联)，以及两个关联量克莱姆 V(cramrv)和关联系数 C(ccc)。
{
    float gammq(float a, float x);
    int nnj,nni,j,i,minij;
    float sum=0.0,expctd,*sumi,*sumj,temp;

    sumi=vector(1,ni);
    sumj=vector(1,nj);
    nni=ni;                             行数
    nnj=nj;                             列数
    for (i=1;i<=ni;i++) {               得到行总数
        sumi[i]=0.0;
        for (j=1;j<=nj;j++) {
            sumi[i] += nn[i][j];
            sum += nn[i][j];
        }
        if (sumi[i] == 0.0) --nni;      通过减少个数消去任何零行
    }
    for (j=1;j<=nj;j++) {               得到列的总数
        sumj[j]=0.0;
        for (i=1;i<=ni;i++) sumj[j] += nn[i][j];
        if (sumj[j] == 0.0) --nnj;      消去任何零列
```

```
    }
    *df=nni*nnj-nni-nnj+1;                          校正的自由度个数
    *chisq=0.0;
    for (i=1;i<=ni;i++) {                           求 χ² 和
        for (j=1;j<=nj;j++) {
            expctd=sumj[j]*sumi[i]/sum;
            temp=nn[i][j]-expctd;
            *chisq += temp*temp/(expctd+TINY);      这里TINY保证消去的行和列不再
        }                                               对和式作出贡献
    }
    *prob=gammq(0.5*(*df),0.5*(*chisq));            χ²概率函数
    minij = nni < nnj ? nni-1 : nnj-1;
    *cramrv=sqrt(*chisq/(sum*minij));
    *ccc=sqrt(*chisq/(*chisq+sum));
    free_vector(sumj,1,nj);
    free_vector(sumi,1,ni);
}
```

### 14.4.2 基于熵的关联测度

考虑"二十个问题"这个游戏。通过重复提问(是/否)问题，除了一种正确的可能性外，能消除所有的未知可能的疑问。考虑一个更一般的游戏，这里你被允许提问双项选择的问题，也可以提问多项选择问题。但多项选择必须是相互排斥的并且是详尽的(正象"是"和"否"一样)。

对你来说，一个答案的值将随所消除的可能性的数目的增加而增加。更特别的是，一个答案如仅提供以分数 $p$ 表征的可能性，那将被赋予值 $-\ln p$(因为 $p<1$，它是一正数)。对数的作用是使这一值呈可加性，因为(例如)一个问题(其仅提供除去 1/6 可能性)将被考虑与二个问题一样好(其以因子 1/2 和 1/3 按顺序将数减少)。

因此这是一个答案的值，但什么是一个问题的值?如果一个问题有 $I$ 种可能的答案($i=1,\ldots,I$)，第 $i$ 个答案的可能性的分数为 $p_i$(所有 $p_i$ 的和等于1)，那么问题的值即为答案值的期望值。记为 $H$

$$H=-\sum_{i=1}^{I}p_i\ln p_i \tag{14.4.6}$$

在计算式(14.4.6)时，应注意

$$\lim_{p\to 0}p\ln p=0 \tag{14.4.7}$$

值 $H$ 在 0 和 $\ln I$ 之间，当仅一个 $p_i$ 等于 1，其它为零时，$H=0$；在这种情况下，问题毫无意义，因为答案是预先规定好的。当所有 $p_i$ 相等时，$H$ 取最大值。在这种情况下问题定然会消去，除剩下可能性的分数 $1/I$ 外的所有可能性。

$H$ 值通常被称为由 $p_i$ 确定的概率分布的**熵**。这一术语出自于统计物理。

迄今为止，我们还未讨论两变量的关联。但假设我们正在决定下面提什么问题和两个候选者的选择，或者可能想按一种顺序或另一种顺序问两个问题。假设一个问题 $x$ 有 $i$ 种可能的答案，每个标记为 $i$。另一个问题 $y$ 有 $J$ 种可能的答案，每个标记为 $j$。那么，对两个问题提问的可能的结果构成了列联表，其每一项为 $N_{ij}$，如用总数 $N$ 归一化，则能给出所有概率 $p$ 的信息，特别是利用式(14.4.1)可以得到

$$p_{ij}=\frac{N_{ij}}{N}$$

$$p_{i\cdot}=\frac{N_{i\cdot}}{N}\qquad \text{（单独问题 } x \text{ 时的结果）}$$

$$p_{\cdot j}=\frac{N_{\cdot j}}{N}\qquad \text{（单独问题 } y \text{ 时的结果）} \tag{14.4.8}$$

问题 $x$ 和 $y$ 的熵分别是：

$$H(x)=-\sum_i p_{i\cdot}\ln p_{i\cdot}\qquad H(y)=-\sum_j p_{\cdot j}\ln p_{\cdot j} \tag{14.4.9}$$

两个问题在一起的熵为（共熵）：

$$H(x,y)=-\sum_{i,j}p_{ij}\ln p_{ij} \tag{14.4.10}$$

什么是给定 $x$ 情况下问题 $y$ 的熵（即如果首先问 $x$），它是位于列联表中某列 $y$ 分布受限的熵（相当于对 $x$ 的答案）在所有 $x$ 上的期望值：

$$H(y|x)=-\sum_i p_{i\cdot}\sum_j\frac{p_{ij}}{p_{i\cdot}}\ln\frac{p_{ij}}{p_{i\cdot}}=-\sum_{i,j}p_{ij}\ln\frac{p_{ij}}{p_{i\cdot}} \tag{14.4.11}$$

反之，则有 $x$ 对 $y$ 的熵

$$H(x|y)=-\sum_j p_{\cdot j}\sum_i\frac{p_{ij}}{p_{\cdot j}}\ln\frac{p_{ij}}{p_{\cdot j}}=-\sum_{i,j}p_{ij}\ln\frac{p_{ij}}{p_{\cdot j}} \tag{14.4.12}$$

我们很容易证明 $y$ 相对 $x$ 的熵将永远不会超过单独 $y$ 的熵，即先问 $x$ 仅能减少提问 $y$ 的有用性（在这种情况下，两变量是相关联的！）

$$\begin{aligned}H(y|x)-H(y)&=-\sum_{i,j}p_{ij}\ln\frac{p_{ij}/p_{i\cdot}}{p_{\cdot j}}\\&=\sum_{i,j}p_{ij}\ln\frac{p_{\cdot j}p_{i\cdot}}{p_{ij}}\\&\le\sum_{i,j}p_{ij}\left(\frac{p_{\cdot j}p_{i\cdot}}{p_{ij}}-1\right)\\&=\sum_{i,j}p_{i\cdot}p_{\cdot j}-\sum_{i,j}p_{ij}\\&=1-1=0\end{aligned} \tag{14.4.13}$$

其中我们使用了以下不等式

$$\ln w\le w-1 \tag{14.4.14}$$

现在，我们已经具有定义 $y$ 对 $x$"依赖性"（即对关联的测度）的一切准备，这一测度有时称为 $y$ 的**不确定系数**，记为 $U(y|x)$

$$U(y|x)=\frac{H(y)-H(y|x)}{H(y)} \tag{14.4.15}$$

这一测度在0和1之间，0值意味着 $x$ 和 $y$ 没有联系，1值意味着对 $x$ 的了解完全能预测 $y$。对任何中间值，$U(x|y)$给出了如果 $x$ 已知，$y$ 的熵 $H(y)$丢失的部分（即对 $x$ 信息的冗余）。在"二十个问题"的游戏中，如果首先提问 $x$，则 $U(y|x)$是问题 $y$ 有用性的部分损失。

如果希望以 $y$ 作为独立变量，$x$ 作为依赖变量，可以通过交换 $x$ 和 $y$ 得到 $x$ 对 $y$ 的依赖性：

$$U(x|y)=\frac{H(x)-H(x|y)}{H(x)} \tag{14.4.16}$$

如果我们希望对称地处理 $x$ 和 $y$，则有用的组合是：

$$U(x,y) \equiv 2\left[\frac{H(y)+H(x)-H(x,y)}{H(x)+H(y)}\right] \tag{14.4.17}$$

如果两个变量完全独立，则 $H(x,y)=H(x)+H(y)$，因此式(14.4.7)等于零。如果两个变量完全依赖，则 $H(x)=H(y)=H(x,y)$，因此式(14.4.7)等于1。实际上，可使用等式(很容易从(14.4.9)~(14.4.12)得到)

$$H(x,y) = H(x) + H(y|x) = H(y) + H(x|y) \tag{14.4.18}$$

并得到：

$$U(x,y) = \frac{H(x)U(x|y) + H(y)U(y|x)}{H(x) + H(y)} \tag{14.4.19}$$

即对称测度是两非对称测度(14.4.15)和(14.4.16)的加权平均，权重分别为每一变量的熵。

以下是一程序，它可以计算所讨论的所有量 $H(x)$、$H(y)$、$H(x|y)$、$H(y|x)$、$H(x,y)$、$U(x|y)$、$U(y|x)$和 $U(x,y)$。

```
#include <math.h>
#include "nrutil.h"
#define TINY 1.0e-30                一个小的数

void cntab2(int **nn, int ni, int nj, float *h, float *hx, float *hy,
    float *hygx, float *hxgy, float *uygx, float *uxgy, float *uxy)
```

给定一个整数型数组nn[i][j]的二维列联表，其中 $i$ 标记 $x$ 变量，其范围从1至 $n_i$；$j$ 标记 $y$ 变量，其范围从1至 $n_j$。本程序返回整个表的熵 h；$x$ 分布的熵 $h_x$，$y$ 分布的熵 $h_y$，已知 $x$ 后 $y$ 的熵 hygx，已知 $y$ 后 $x$ 的熵 hxgy，$y$ 关于 $x$ 的依赖性 uygx 为式(14.4.15)，$x$ 关于 $y$ 的依赖性 uxgy 为式(14.4.16)，对称依赖性 uxy 为式(14.4.17)。

```
{
    int i,j;
    float sum=0.0,p,*sumi,*sumj;

    sumi=vector(1,ni);
    sumj=vector(1,nj);
    for (i=1;i<=ni;i++) {                   得到行总数
        sumi[i]=0.0;
        for (j=1;j<=nj;j++) {
            sumi[i] += nn[i][j];
            sum += nn[i][j];
        }
    }
    for (j=1;j<=nj;j++) {                   得到列总数
        sumj[j]=0.0;
        for (i=1;i<=ni;i++)
            sumj[j] += nn[i][j];
    }
    *hx=0.0;                                x分布的熵
    for (i=1;i<=ni;i++)
        if (sumi[i]) {
            p=sumi[i]/sum;
            *hx -= p*log(p);
        }
    *hy=0.0;                                y分布的熵
    for (j=1;j<=nj;j++)
        if (sumj[j]) {
            p=sumj[j]/sum;
            *hy -= p*log(p);
        }
    *h=0.0;
    for (i=1;i<=ni;i++)                     总熵：关于x和y的循环
        for (j=1;j<=nj;j++)
```

```
            if (nn[i][j]) {
                p=nn[i][j]/sum;
                *h -= p*log(p);
            }
    *hygx=(*h)-(*hx);                                  用(14.4.18)式
    *hxgy=(*h)-(*hy);
    *uygx=(*hy-*hygx)/(*hy+TINY);                      求(14.4.15)式
    *uxgy=(*hx-*hxgy)/(*hx+TINY);                      求(14.4.16)式
    *uxy=2.0*(*hx+*hy-*h)/(*hx+*hy+TINY);              求(14.4.17)式
    free_vector(sumj,1,nj);
    free_vector(sumi,1,ni);
}
```

参考文献和进一步读物：


Dunn, O. J., and Clark, V. A. 1974, *Applied Statistics: Analysis of Variance and Regression* (New York: Wiley).

Fano, R. M. 1961, *Transmission of Information*(New York: Wiley and MIT Press), Chapter 2.


## 14.5 线性相关

我们下面将继续讨论两变量之间关联的测度，这两个变量是顺序量和连续量，而不是公称量。最常用的是**线性相关系数**。对数据对$(x_i,y_i)$，$i=1,\ldots,N$，线性相关系数$r$(也称为乘积矩相关系数，或皮尔逊(Pearson) $r$)定义为：

$$r=\frac{\sum_i (x_i-\bar{x})(y_i-\bar{y})}{\sqrt{\sum_i (x_i-\bar{x})^2}\sqrt{\sum_i (y_i-\bar{y})^2}} \tag{14.5.1}$$

其中$\bar{x}$、$\bar{y}$分别是$\{x_i\}$和$\{y_i\}$的均值。

$r$的值位于$-1$和$1$之间，包括$-1$和$1$。当数据点完全位于一条正斜率的直线上($x$和$y$一起增加)时，$r$取值为1，称为完全正相关。当完全位于一条负斜率的直线上($x$增加而$y$减少)则$r$取$-1$，它被称为完全负相关。$r$接近0表明变量$x$和$y$量是不相关的。

当已知一个相关是显著的，则$r$是测度强度的惯用方式，实际上，$r$的值等价于以下表述，即当数据利用最小二乘方拟合成一条直线时(见第15.2节特别是(15.2.13)、(15.2.14))，什么样的残差(偏差的均方根)是所期望的呢?遗憾的是，对于确定一个观察到的相关性是否在统计上是显著的，或者一个观察到的相关性是否显著地比另一个强时，$r$是个相当差的统计量，其原因是$r$对$x$和$y$的单独分布是未知的，因此在无效假设情况下，没有任何一般的方式可计算其分布。

仅能做的工作是：如果无效假设为$x$和$y$是不相关的，而且如果关于$x$和$y$都有足够的收敛矩(其末端下降很快)，又如果$N$足够大(典型>500)，则$r$近似为正态分布，均值为0，方差为$1/\sqrt{N}$。在这种情况下，相关的(双边)显著性，即无效假设下，$|r|$大于观察的值概率，为

$$\mathrm{erfc}\left(\frac{|r|\sqrt{N}}{\sqrt{2}}\right) \tag{14.5.2}$$

其中$\mathrm{erfc}(x)$是式(6.2.8)表达的补余误差函数，它由第6.2节的程序**erffc**或**erfcc**计算。式(14.5.2)的较小值表明两分布是显著相关的(下面式(14.5.9)是更精确的检验)。

许多统计学教科书都试图超出式(14.5.2)的范围，并给出可以利用$r$的另外的统计检

验。然而，在几乎所有的情况下，这些检验仅对一些特殊的假设有效，即 $x$ 和 $y$ 的联合分布形成了一个双正态或二维高斯分布，联合概率分布为：

$$p(x,y)dxdy = 常数 \times \exp\left[-\frac{1}{2}(a_{11}x^2 - 2a_{12}xy + a_{22}y^2)\right]dxdy \tag{14.5.3}$$

其中，$a_{11}$、$a_{12}$、$a_{22}$是任意常数，对这一分布 $r$ 取值为

$$r = -\frac{a_{12}}{\sqrt{a_{11}a_{22}}} \tag{14.5.4}$$

有时式(14.5.3)被看成一个好的数据模型，有时至少被看作粗糙而又便利的假设，因为许多两维分布相似于双正态分布，至少在尾端相差不多。在两种情况下，我们能在以下几个方向超出式(14.5.2)的范围来使用式(14.5.6)：

首先，我们能考虑数据点数目 N 不很大时的可能情况，这里统计量

$$t = r\sqrt{\frac{N-2}{1-r^2}} \tag{14.5.5}$$

在无效假设情况(不相关)下的分布与 $\nu=N-2$自由度的学生 $t$ 分布相像，其双边显著性水平由$1-A(t/\nu)$(方程(6.4.7))给出。当 $N$ 增大时，这显著性将逐次变得与式(14.5.2)相同。因此人们使用式(14.5.5)将不会变得更糟，既使双正态假设还未被很好地证明。

其次，当 $N$ 仅是中等大小($\geqslant 10$)时，我们能够比较是否两个显著非零 $r$ 值之间的差异(例如从不同的实验)本身是显著的。换句话说，我们能定量地确定，在某一控制变量上的变化是否能显著地改变两个其它变量之间所存在的相关性，做到这点是通过利用 Fisher 的 $z$ 变换将每个测量的 $r$ 与相应的量 $z$ 联系起来

$$z = \frac{1}{2}\ln\left(\frac{1+r}{1-r}\right) \tag{14.5.6}$$

于是，每个 $z$ 都近似为正态分布，其均值为

$$\bar{z} = \frac{1}{2}\left[\ln\left(\frac{1+r_{true}}{1-r_{true}}\right) + \frac{r_{true}}{N-1}\right] \tag{14.5.7}$$

其中 $r_{true}$是相关系数的实际值或总体值，并且具有以下标准差

$$\sigma(z) \approx \frac{1}{\sqrt{N-3}} \tag{14.5.8}$$

当式(14.5.7)和(14.5.8)有效时，给出了几个有用的统计检验。例如 $r$ 的测量值不同于某个假设值 $r_{true}$的显著性水平为：

$$\text{erfc}\left(\frac{|z-\bar{z}|\sqrt{N-3}}{\sqrt{2}}\right) \tag{14.5.9}$$

其中 $z$ 和 $\bar{z}$ 由式(14.5.6)和(14.5.7)给出，并且式(14.5.9)的值较小表示有显著差别(令 $\bar{z}=0$，用式(14.5.9)代替(14.9.2)会更精确)。与此相似，两个测量的相关系数 $r_1$和 $r_2$之间差异的显著性为：

$$\text{erfc}\left(\frac{|z_1 - z_2|}{\sqrt{2}\sqrt{\frac{1}{N_1-3}+\frac{1}{N_2-3}}}\right) \tag{14.5.10}$$

其中 $z_1$和 $z_2$可以利用式(14.5.6)由 $r_1$和 $r_2$得到，$N_1$和 $N_2$分别是 $r_1$和 $r_2$的度量中数据点的数

目。

所有上面的显著性都是双边的。如果希望否定无效假设以有利于单边假设，例如 $r_1 > r_2$（其中不等式的估计方向被先验地决定了），那么：(i)如果测度的 $r_1$ 和 $r_2$ 有错误约估计方向，则将不能证明所作的单边假设；但(ii)如果它们有正确的次序，则能够用 0.5 乘以上面给出的各种显著性，就使得它们更加显著。

但请记住：如果 $x$ 和 $y$ 变量的联合概率分布是显著不同于于双正态分布，则 $r$ 统计量的解释可能完全无意义。

```
#include <math.h>
#define TINY 1.0e-20                                将正则化完全相关的异常情况

void pearsn(float x[], float y[], unsigned long n, float *r,
    float *prob, float *z)
    给定两个数组x[1..n]和y[1..n]，本程序计算它们的相关系数r(返回为r)，零相关的无效假设被否定的显著性
    水平(prob，它的小值表示有显著相关)，以及Fisher z(返回z)。它的值可以用于如上所述的进一步统计检验中
{
    float betai(float a, float b, float x);
    float erfcc(float x);
    unsigned long j;
    float yt,xt,t,df;
    float syy=0.0,sxy=0.0,sxx=0.0,ay=0.0,ax=0.0;

    for (j=1;j<=n;j++) {                                                    寻找均值
        ax += x[j];
        ay += y[j];
    }
    ax /= n;
    ay /= n;
    for (j=1;j<=n;j++) {                                                    计算相关系数
        xt=x[j]-ax;
        yt=y[j]-ay;
        sxx += xt*xt;
        syy += yt*yt;
        sxy += xt*yt;
    }
    *r=sxy/sqrt(sxx*syy);
    *z=0.5*log((1.0+(*r)+TINY)/(1.0-(*r)+TINY));                          Fisher z 变换
    df=n-2;
    t=(*r)*sqrt(df/((1.0-(*r)+TINY)*(1.0+(*r)+TINY)));                     式(14.5.5)
    *prob=betai(0.5*df,0.5,df/(df+t*t));                                   学生t概率
    /* *prob=erfcc(fabs((*z)*sqrt(n-1.0))/1.4142136) */                    对于大n，使用程序erfcc，它
                                                                           较易计算概率，并给出近似
                                                                           相同的值
}
```

## 14.6 非参数相关或秩相关

正是在线性相关系数 $r$ 的显著性解释中的不定性，导致了**非参数相关或秩相关**这些重要概念。象以前一样，给出 $N$ 对测量值$(x_i, y_i)$。以前的困难在于我们未必知道$\{x_i\}$和$\{y_i\}$母体的概率分布函数。

非参数相关的关键概念是：在所有样本$\{x_i\}$中，如果我们用每一个 $x_i$ 的秩值来代替 $x_i$，即 1,2,3,…,$N$，则作为结果的数表将取样于完全已知的分布函数，即均匀地来自从 1 到 $N$

(包括 1 和 $N$)的整数。实事上,这比均匀更好,因为如果 $x_i$ 的值都是不同的,每个整数都将只出现一次。如果$\{x_i\}$中有相等的,通常赋于这些"相持"以秩的均值(如果它们有稍稍不同,就会有此均值)。这些中间秩有时是整数,有时是半整数。在所有情况下,所有赋于秩的和都将等于从1到 $N$ 所有整数的和,即$\frac{1}{2}N(N+1)$。

当然我们可以对$\{y_i\}$进行同样的处理,即用每个值在样本中的秩代替每个 $y_i$。

现在,我们可以自由地选择监测均匀整数组(1..$N$)之间相关性的统计量,应注意秩中"相持"的可能性。当然在用秩代替原始值时会失掉一些信息。我们能构造一些相当人为的例子,其相关性能够参数化地(如以线性相关系数 $r$)检测,但并不能非参数化地检测。然而,这样的例子在实际生活中是很少见的,并且在秩化过程中,稍微的信息损失是以很小的代价换取高的收益;当一相关性被证明是非参数的,那么它确实如此!(即一定程度上取决于选择的显著性。)非参数相关比线性相关更稳健,也更宜于消除数据中未知的缺陷,就象中位数比均值更稳健一样。关于稳健性更详细的讨论见第15.7节。

在统计学中总是有些已构造好的统计量为我所用。下面将讨论两个统计量,Spearman 秩阶相关系数($r_s$)和 Kendall 的 $tau(\tau)$。

### 14.6.1 Spearman 秩阶相关系数

设 $R_i$ 为$\{x_i\}$中 $x_i$ 的秩,$S_i$ 为$\{y_i\}$中 $y_i$ 的秩,并也被赋于中间秩,则秩阶相关系数被定义为秩的线性相关系数,即

$$r_s = \frac{\sum_i (R_i - \bar{R})(S_i - \bar{S})}{\sqrt{\sum_i (R_i - \bar{R})^2}\sqrt{\sum_i (S_i - \bar{S})^2}} \tag{14.6.1}$$

检验非零值 $r_s$ 的显著性由下式给出:

$$t = r_s\sqrt{\frac{N-2}{1-r_s^2}} \tag{14.6.2}$$

其近似为 $N-2$自由度的学生分布。关键是这一近似并不取决于 $x$ 和 $y$ 的原始分布;它始终是同样的近似,同样地好。

可以证明,$r_s$ 与另一种常用的非参数相关的测度密切相关,即秩差的平方和:

$$D = \sum_{i=1}^{N}(R_i - S_i)^2 \tag{14.6.3}$$

($D$ 有时用 $D^{**}$ 表示,其中 * 号表明相持是用中间秩处理的。)

当数据中没有相持时,则 $D$ 和 $r_s$ 的关系为:

$$r_s = 1 - \frac{6D}{N^3 - N} \tag{14.6.4}$$

当有相持时,则关系更复杂一些:设 $f_k$ 为$\{R_i\}$的第 $k$ 个相持组中相持的数目,$g_m$ 为$\{S_i\}$的第 $m$ 个相持组中相持的数目,则得出下式精确地成立:

$$r_s = \frac{1 - \frac{6}{N^3 - N}\left[D + \frac{1}{12}\sum_k (f_k^3 - f_k) + \frac{1}{12}\sum_m (g_m^3 - g_m)\right]}{\left[1 - \frac{\sum_k (f_k^3 - f_k)}{N^3 - N}\right]^{1/2}\left[1 - \frac{\sum_m (g_m^3 - g_m)}{N^3 - N}\right]^{1/2}} \tag{14.6.5}$$

注意，如所有 $f_k$ 和 $g_m$ 都等于 1（即没有相持），则式(14.6.5)约化为式(14.6.4)。

在式(14.6.2)中我们给出了检验非零 $r_s$ 显著性的 $t$ 统计量。而用它直接检验 $D$ 的显著性也是可行的，在无效假设（非相关数据组）中 $D$ 的期望值为：

$$\overline{D} = \frac{1}{6}(N^3 - N) - \frac{1}{12}\sum_k(f_k^3 - f_k) - \frac{1}{12}\sum_m(g_m^3 - g_m) \qquad (14.6.6)$$

其方差为：

$$\mathrm{Var}(D) = \frac{(N-1)N^2(N+1)^2}{36} \times \left[1 - \frac{\sum_k(f_k^3 - f_k)}{N^3 - N}\right]\left[1 - \frac{\sum_m(g_m^3 - g_m)}{N^3 - N}\right] \qquad (14.6.7)$$

并且，它近似服从正态分布。因此显著性水平为补余误差函数（见式(14.7.2)）。当然，式(14.6.2)和(14.6.7)都不独立于检验，而是同一检验的简单变形。在下面的程序中我们将同时计算由式(14.6.2)和(14.6.7)给出的显著性水平；它们的差异将给出这样的概念：近似好到什么程度。还将注意到，我们将指派秩的工作（包括相持的中间秩）分出来由子程序 crank 完成。

```
#include <math.h>
#include "nrutil.h"

void spear(float data1[], float data2[], unsigned long n, float *d,
    float *zd, float *probd, float *rs, float *probrs)
    给定两个数据数组 data1[1..n] 和 data2[1..n]，程序将它们的秩差之平方和返回到 D，D 偏差无效假设期望值的标
    准差为 zd，这种偏差的双边显著性水平为 probd，Spearman 秩相关 rs 返回为 rs，它偏离零双边显著性水平为 pro
    brs。外部程序 crank（下面）和 sort2（第8.2）被使用，无论小的 probd 和小的 probrs 值都表示一个显著地相关（rs
    为正）或者反相关（rs 为负）。
{
    float betai(float a, float b, float x);
    void crank(unsigned long n, float w[], float *s);
    float erfcc(float x);
    void sort2(unsigned long n, float arr[], float brr[]);
    unsigned long j;
    float vard,t,sg,sf,fac,en3n,en,df,aved,*wksp1,*wksp2;

    wksp1=vector(1,n);
    wksp2=vector(1,n);
    for (j=1;j<=n;j++) {
        wksp1[j]=data1[j];
        wksp2[j]=data2[j];
    }
    srot2(n,wksp1,wksp2);                  对每一数组排序，并将表项转换成秩。sf 和 sg 的值
    crank(n,wksp1,&sf);                    分别返回 ∑(f_k^3 − f_k) 和 ∑(g_m^3 − g_m) 的值
    sort2(n,wksp2,wksp1);
    crank(n,wksp2,&sg);
    *d=0.0;
    for (j=1;j<=n;j++)
        *d += SQR(wksp1[j]-wksp2[j]);
    en=n;
    en3n=en*en*en-en;
    aved=en3n/6.0-(sf+sg)/12.0;                            D 的期望值
    fac=(1.0-sf/en3n)*(1.0-sg/en3n);
    vard=((en-1.0)*en*en*SQR(en+1.0)/36.0)*fac;            D 的方差给出
```

```
    *zd=(*d-aved)/sqrt(vard);                                   标准差的个数和显著性
    *probd=erfcc(fabs(*zd)/1.4142136);
    *rs=(1.0-(6.0/en3n)*(*d-(sf+sg)/12.0))/sqrt(fac);           秩相关系数
    fac=(*rs+1.0)*(1.0-(*rs));
    if (fac > 0.0) {
        t=(*rs)*sqrt((en-2.0)/fac);                             和它的t值
        df=en-2.0;
        *probrs=betai(0.5*df,0.5,df/(df+t*t));                  给出它们的显著性
    } else
        *probrs=0.0;
    free_vector(wksp2,1,n);
    free_vector(wksp1,1,n);
}

void crank(unsigned long n, float w[], float *s)
```

给定一个已排序的数组 w[1..n]，用它们的秩，包括相持的中间秩来代替元素，以及将 $f^3-f$ 的和式返回到 $s$，其中 $f$ 是每个相持中元素的个数。

```
{
    unsigned long j=1,ji,jt;
    float t,rank;

    *s=0.0;
    while (j < n) {
        if (w[j+1] != w[j]) {                                   不是相持
            w[j]=j;
            ++j;
        } else {                                                有相持
            for (jt=j+1;jt<=n && w[jt] == w[j];jt++);           它走了多远
            rank=0.5*(j+jt-1);                                  这是相持的平均秩
            for (ji=j;ji<=(jt-1);ji++) w[ji]=rank;              将此平均秩送入到所有的相持表项中
            t=jt-j;                                             并修正s
            *s += t*t*t-t;
            j=jt;
        }
    }
    if (j == n) w[n]=n;                                         如果最后元素不是相持，这就是它的秩
}
```

## 14.6.2 Kendall 的 $\tau$

肯德尔(Kendall)的 $\tau$ 甚至比 Spearman 的 $r_s$ 或 $D$ 更非参数化。它将不用秩的数值差，而只和秩的相对次序关系：较高秩、较低秩、同秩。在这种情况下，我们甚至不必求秩，仅需通过值的大、小、相等就可分别确定秩的较高、较低、相等。总的说来，我们更喜欢 $r_s$ 为直接的非参数检验，但是这两个统计量都很常用。事实上，$\tau$ 和 $r_s$ 是强相关，并且在多数应用中是同样有效的检验。

为了定义 $\tau$，以 $N$ 个数据点 $(x_i, y_i)$ 作为讨论的起点。现在考虑所有 $\frac{1}{2}N(N-1)$ 对数据点，其中数据点自身不能成对，但按任一种顺序计数的点都可作为一对。如果两个 $x$ 秩的相对次序关系与两个 $y$ 秩的相对次序关系相同则称该对和谐的，如果次序关系相反，则称为不和谐的。如果在 $x$ 或 $y$ 中有一相持，则我们既不能把点对称为和谐的，也不能称为不和谐的。如果相持在 $x$ 中，则称该点对为"附加的 $y$"对；如果相持在 $y$ 中，则称该点对为"附加的 $x$"对。如果 $x$、$y$ 同时有相持，我们根本不能对该点对称呼什么，读者和我们的观点一致吗？

Kendall 的 $\tau$ 是这些量的简单组合：

$$\tau = \frac{\text{和谐量} - \text{不和谐量}}{\sqrt{\text{和谐量} - \text{不和谐量} + \text{附加}\, y\, \text{的量}}\sqrt{\text{和谐量} + \text{不和谐量} + \text{附加}\, x\, \text{的量}}} \tag{14.6.8}$$

显然，$\tau$ 在 $-1$ 和 $+1$ 之间，并且仅在秩完全一致或相反时才能取得极值。

更重要的是，肯德尔已经由组合算出，$x$ 和 $y$ 之间没有联系的无效假设下 $\tau$ 的近似分布。在这种情况下，$\tau$ 近似为正态分布，期望值为 0，方差为：

$$\mathrm{Var}(\tau) = \frac{4N+10}{9N(N-1)} \tag{14.6.9}$$

以上描述可归结为下面的程序。应该知道：这是一个 $O(N^2)$ 的算法，它不象 $r_s$ 算法，其主要的排序运算是 $N\log N$ 阶。如果按常规对超过几千个点的数据组计算 Kendall $\tau$ 值，则将非常麻烦。然而，如果愿意将数据合并成中等数目的离散值，则可用下述程序：

```
#include <math.h>

void kendl1(float data1[], float data2[], unsigned long n, float *tau,
    float *z, float *prob)
    给定数组 data1[1..n]和 data2[1..n]，本程序将肯德尔 τ 返回到 tau，将它的偏离零点标准差返回到 z，并将它的双
    边显著性水平返回到 prob。小值的 prob 表示显著地相关(tau 为正)或者反相关(tau 为负)。
{
    float erfcc(float x);
    unsigned long n2=0,n1=0,k,j;
    long is=0;
    float svar,aa,a2,a1;

    for (j=1;j<n;j++) {                              对点对的第一个元素循环
        for (k=(j+1);k<=n;k++) {                     对第二个元素循环
            a1=data1[j]-data1[k];
            a2=data2[j]-data2[k];
            aa=a1*a2;
            if (aa) {                                两个数值都没有相持
                ++n1;
                ++n2;
                aa > 0.0 ? ++is : --is;
            } else {                                 一个或两个数组有相持
                if (a1) ++n1;                        一个"附加的 x"事件
                if (a2) ++n2;                        一个"附加的 y"事件
            }
        }
    }
    *tau=is/(sqrt((double) n1)*sqrt((double) n2));   方程(14.6.8)式
    svar=(4.0*n+10.0)/(9.0*n*(n-1.0));               方程(14.6.9)式
    *z=(*tau)/sqrt(svar);
    *prob=erfcc(fabs(*z)/1.4142136);                 显著性
}
```

有时，恰巧对每个 $x$ 和 $y$ 只有几个可能的值。在那种情况下，数据能被记录成列联表（见第14.4节），该表对每一 $x$ 和 $y$ 对的列联给出了数据点数目。

Spearman 的秩阶相关系数在这些情况下并不是非常自然的统计量，因为它赋予每个 $x$ 和 $y$ 量化级并不非常有意义的中间秩值，这导致了大量的等秩差。换言之，Kendall $\tau$ 由于计算简单，可以相当合乎实际。因此，它的 $O(N^2)$ 算法再也不是问题，因为我们可以安排它在成对的列联表的项目上（每个包含许多数据点）进行循环，而不是对成对的数据点进行循环。这点将在下面的程序中加以改进。

注意，Kendall $\tau$ 只能用于两变量是非常有序的（即良序的）列联表，并且它只寻求单调

相关,而不是任意的关联。这两点使它与第14.4的方法相比减少了普遍性,第14.4的方法可应用于**公称**(即无序的)变量的任意关联。

比较上面的 kendl1 和下面的 kendl2,将看到我们已对许多变量"浮点化"。这是因为在列联表中事件的数目相当大,以致在一些整型算式中会出现溢出,而在表中单个数据点的数目都是不可能那么大(对一个 $O(N^2)$ 程序而言)!

```
#include <math.h>

void kendl2(float **tab, int i, int j, float *tau, float *z, float *prob)
    给定一个二维表 tab[1..i][1..j]使 tab[k][l]包含一个变量落入第 k 个量化级,第二个变量落入第 l 个量化级的事
    件的个数。本程序将 Kendall τ 返回到 tau,将它偏离零点的标准差返回到 z,并将它的双边显著水平返回到 prob。小
    值的 prob 表示两变量之间显著相关(tau 为正)或者反相关(tau 为负)。虽然 tab 是浮点型数组,但通常它是整数值
{
    float erfcc(float x);
    long nn,mm,m2,m1,lj,li,l,kj,ki,k;
    float svar,s=0.0,points,pairs,en2=0.0,en1=0.0;

    nn=i*j;                                    列联表中表项的总数
    points=tab[i][j];
    for (k=0;k<=nn-2;k++) {                    对表中表项进行循环
        ki=(k/j);                              对一行和一列
        kj=k-j*ki;                             进行译码
        points += tab[ki+1][kj+1];             增加事件的总数
        for (l=k+1;l<=nn-1;l++) {              对点对的其它成员进行循环
            li=l/j;                            对它的行和列
            lj=l-j*li;                         进行译码
            mm=(m1=li-ki)*(m2=lj-kj);
            pairs=tab[ki+1][kj+1]*tab[li+1][lj+1];
            if (mm) {                          不是相持
                en1 += pairs;
                en2 += pairs;
                s += (mm > 0 ? pairs : -pairs);         和谐的或不合谐的
            } else {
                if (m1) en1 += pairs;
                if (m2) en2 += pairs;
            }
        }
    }
    *tau=s/sqrt(en1*en2);
    svar=(4.0*points+10.0)/(9.0*points*(points-1.0));
    *z=(*tau)/sqrt(svar);
    *prob=erfcc(fabs(*z)/1.4142136);
}
```

## 14.7 二维分布不同吗?

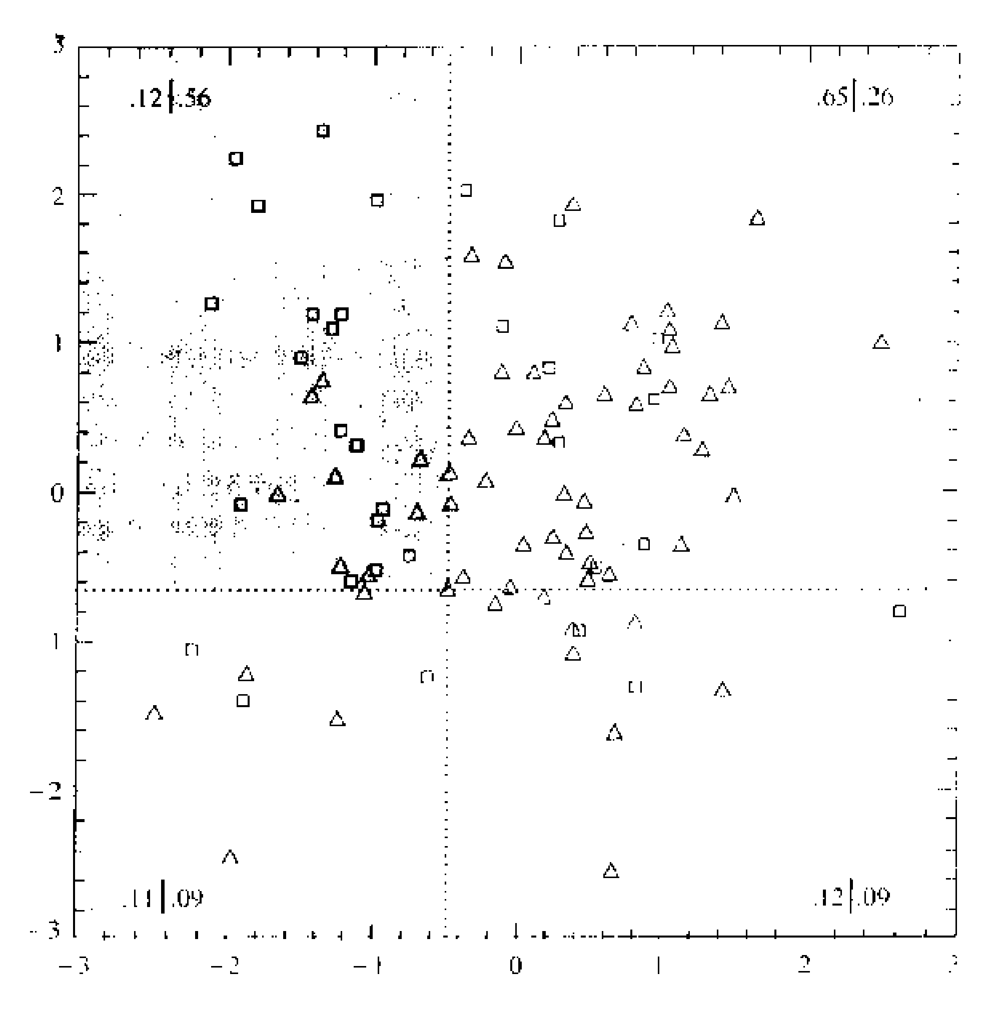
下面我们讨论把 K-S 检验推广到二维分布。这一推广的过程应归功于 Fasano 和 Franceschini[1],实际上它是 Peacock[2] 早期概念的变异。

在二维分布中,每个数据点都用一对值 $(x,y)$ 来表征。一个我们熟悉的例子是,所探测到的来自超新星 1987A 的 19 个中微子,它们每一个都由时间 $t_i$ 和能量 $E_i$ 来表征(见[3])。我们希望知道这些测量的数据对 $(t_i,E_i)$,$i=1...19$ 是否与理论模型一致,理论模型预期中微子流是时间和能量的函数,即 $(x,y)$ 平面(这里是 $(t,E)$ 平面)的二维概率分布。这是单样本检验。如果给出来自两个可比较的观测仪的两组中微子探测结果,我们希望知道这两者是否互相一致,即这是双样本检验。

我们希望将一维 K-S 检验引入 $(x,y)$ 平面,以寻求两个二维分布之间的某种最大**积累差**。遗憾的是,在

超过一维的情况下，不容易定义积累的概率分布！Peacock 的贡献是找到了一种好的替代物，即在给定点 $(x_i, y_i)$ 周围**四个自然象限每一个象限中的综合概率**，亦即在 $(x>x_i, y>y_i)$、$(x<y_i, y>y_i)$、$(x<x_i, y<y_i)$、$(x>x_i, y<y_i)$ 中的总概率（或数据的分数）。二维 K-S 统计量 $D$ 现在可看成相应于综合概率的最大差异（它涉及数据点和象限两个范围）。当比较两个数据集时，$D$ 值取决于数据集变化的范围。在这种情况下，定义一个有效的 $D$ 为所得到两个值的平均，如果对 $D$ 定义感到困惑，不要发愁，相应的计算机程序会提供精确的算法定义。

图14.7.1给出了一个例子，在平面中 65 个三角形和 35 个正方形似乎具有不同的分布，点划线的中心位于使统计量 $D$ 最大的三角形上；最大值出现在左上象限，在这象限中包含了只有 12% 的三角形，但包含了 56% 的正方形，所以 $D$ 值为 0.44，这是否在统计上显著不同呢？




二维 K-S 检验找到了这样的点，其象限之一（由点划线给出）内三角形分数和正方形分数之间的差异最大。那么式(14.7.1)指出了这一差异是否在统计上是显著的，即三角形和正方形是否依从不同的分布。

图14.7.1　65个三角形和35个正方形的二维分布

遗憾的是，甚至对固定的样本大小，下面陈述也不是严格正确的：在无效假设下，$D$ 的分布独立于二维分布的形状。在这方面，二维 K-S 检验并不象一维那样实用。然而，使用广义的蒙特卡罗积分已表明，即使对非常不同的分布，二维 $D$ 的分布也极近似相同的，只要它们有相同的相关系数 $r$（是以式(14.5.1)相同的公式定义的）。Fasano 和 Franceschini 在他们的文章中列出了 $D$ 分布作为 $D$、样本大小 $N$ 和相关系数 $r$ 的函数的蒙特卡罗结果。分析这些结果，人们发现两维 K-S 检验的显著性水平由下面一个简单但又近似的公

式给出：

对单样本检验，

$$\text{概率}(D > \text{观察值}) = Q_{KS}\left(\frac{\sqrt{N}D}{1+\sqrt{1-r^2}(0.25-0.75/\sqrt{N})}\right) \tag{14.7.1}$$

对双样本检验，上面公式仍适用，只是有：

$$N = \frac{N_1N_2}{N_1+N_2} \tag{14.7.2}$$

当 $N \gtrsim 20$ 且所指示的概率(显著性水平)小于(更加显著) 0.20 左右时，上面公式是足够精确的。如果所指示的概率大于 0.2，则其值不再精确，但也意味着数据和模型(或两数据组)并不显著地不同，这一点是正确的。注意，当 $r \to 1$(完全相关)时，式(14.7.1)和(14.7.2)归结到式(14.3.9)和(14.3.10)：二维数据全位于直线上，并且二维 K-S 检验变成一维 K-S 检验。

顺便说一下，图14.7.1的显著性水平是 0.001 左右。这就更加明确三角形和正方形取自不同的分布(实际上也是如此)。

当然，如果不想依靠体现在式(14.7.1)中的蒙特卡罗实验，那么可以寻求自己的方式：从模型中造出许多假想数据集，每个集中的数据点都与真实数据集相同，对每一假想集，利用相应的计算程序计算 $D$(但忽略它们的计算概率)，数出这些假想 $D$ 超过真实 $D$(来自实际数据组)的次数的百分比，这个分数就是所需的显著性。

与一维检验比较，二维检验的缺点是其需要 $N^2$ 次运算，两个 $N$ 阶嵌套循环占据了 $N\log N$ 次操作。对小型计算机来说，这使得检验只限于 $N$ 小于几千的范围内才有用。

现在我们给出计算机执行程序。一个样本的情况包含在 ks2d1s(即二维，1个样本)。这个程序调用一个直接可用的程序 **quadct** 来计算四个象限中的点，并且调用了一个用户应准备的程序 **quadvl**，其能给出任意一点 $(x,y)$ 附近四个象限中的每一个象限的综合概率。下面也给出一个不重要的示例 **quadvl**，而真实的 **quadvl** 是相当复杂的，并经常伴随解析二维分布的数值积分。

```
#include <math.h>
#include "nrutil.h"

void ks2d1s(float x1[], float y1[], unsigned long n1,
    void (*quadvl)(float, float, float *, float *, float *, float *), float *d1, float *prob)
    一个样本对于一个模型的K-S检验。在数组x1[1..n1]和y1[1..n1]中给定n1个数据点的x和y的坐标，以及给定
    一个用户提供的并作为模型例子的函数quadvl，本程序返回二维K-S统计到d1，它的显著性水平返回到prob。小值
    的prob表明样本显著地不同于模型。注意这种检验直接依赖于分布，所以prob只是一个估算值。
{
    void pearsn(float x[], float y[], unsigned long n, float *r, float *prob, float *z);
    float probks(float alam);
    void quadct(float x, float y, float xx[], float yy[], unsigned long nn,
        float *fa, float *fb, float *fc, float *fd);
    unsigned long j;
    float dum,dumm,fa,fb,fc,fd,ga,gb,gc,gd,r1,rr,sqen;

    *d1=0.0;
    for (j=1;j<=n1;j++) {                          对所有数据点循环
        quadct(x1[j],y1[j],x1,y1,n1,&fa,&fb,&fc,&fd);
        (*quadvnl)(x1[j],y1[j],&ga,&gb,&gc,&gd);
        *d1=FMAX(*d1,fabs(fa-ga));
        *d1=FMAX(*d1,fabs(fb-gb));
        *d1=FMAX(*d1,fabs(fc-gc));
        *d1=FMAX(*d1,fabs(fd-gd));         对样本和模型，累积每一个象限中的分布，并且存储最大差
    }
    pearsn(x1,y1,n1,&r1,&dum,&dumm);         求得线性相关系数r1
```

```
    sqen=sqrt((double)n1);
    rr=sqrt(1.0-r1*r1);
    *prob=probks(*d1*sqen/(1.0+rr*(0.25-0.75/sqen)));
```

用 K-S 概率函数 probks 来估算概率值

```
}

void quadct(float x, float y, float xx[], float yy[], unsigned long nn,
    float *fa, float *fb float *fc, float *fd)
```

给定原点$(x,y)$并给定 nn 点数组其坐标为 xx[1..nn]和 yy[1..nn]，计算围绕原点的每个象限中有多少数据点，并将归一化的分数返回，每个象限用英文字母为下标标记，从右上角逆时针方向计算起，此程序用于 ks2d1s 和 ks2d2s 中。

```
{
    unsigned long k,na,nb,nc,nd;
    float ff;
    na=nb=nc=nd=0;
    for (k=1;k<=nn;k++) {
        if (yy[k] > y) {
            xx[k] > x ? ++na : ++nb;
        } else {
            xx[k] > x ? ++nd : ++nc;
        }
    }
    ff=1.0/nn;
    *fa=ff*na;
    *fb=ff*nb;
    *fc=ff*nc;
    *fd=ff*nd;
}

#include "nrutil.h"

void quadvl(float x, float y, float *fa, float *fb, float *fc, float *fd)
```

这是被用于 ks2d1s 中用户所提供的程序样本，模型的分布是均匀的在$-1<x<1,-1<y<1$区间以内。一般情况下，本程序应该对每点(x,y)返回围绕此点四个象限之每一象限中，总分布的分数 fa、fb、fc 和 fd 相加必须为1，每个象限用英文字母为下标标记，从右上角逆时针方向计算。

```
{
    float qa,qb,qc,qd;

    qa=FMIN(2.0,FMAX(0.0,1.0-x));
    qb=FMIN(2.0,FMAX(0.0,1.0-y));
    qc=FMIN(2.0,FMAX(0.0,x+1.0));
    qd=FMIN(2.0,FMAX(0.0,y+1.0));
    *fa=0.25*qa*qb;
    *fb=0.25*qb*qc;
    *fc=0.25*qc*qd;
    *fd=0.25*qd*qa;
}
```

程序 **ks2d2s** 是两个样本的二维 K-S 检验。它也调用 **quadct**、**pearsn** 和 **probks**。进行两个样本检验时，不需要解析的模型。

```
#include <math.h>
#include "nrutil.h"

void ks2d2s(float x1[], float y1[], unsigned long n1, float x2[], float y2[],
    unsigned long n2, float *d, float *prob)
```

两个样本的二维 K-S 检验。给定数组 x1[1..n1]和 y1[1..n1]为第一个样本 n1个值的 x、y 坐标。同样，数组 x2和 y2为第二个样本 n2的 x、y 坐标。本程序将二维、两个样本的 K-S 统计量返回为 d，它的显著性水平为 prob。小值的

prob 表明这两个样本显著不同，注意，这种检验直接依赖于分布，所以，prob 只是一种估算量。

```
{
    void pearsn(float x[], float y[], unsigned long n, float *r, float *prob, float *z);
    float probks(float alam);
    void quadct(float x, float y, float xx[], float yy[], unsigned long nn,
        float *fa, float *fb, float *fc, float *fd);
    unsigned long j;
    float d1,d2,dum,dumm,fa,fb,fc,fd,ga,gb,gc,gd,r1,r2,rr,sqen;

    d1=0.0;
    for (j=1;j<=n1;j++) {                    首先，用第一样本中的点作为原点
        quadct(x1[j],y1[j],x1,y1,n1,&fa,&fb,&fc,&fd);     gins.
        quadct(x1[j],y1[j],x2,y2,n2,&ga,&gb,&gc,&gd);
        d1=FMAX(d1,fabs(fa-ga));
        d1=FMAX(d1,fabs(fb-gb));
        d1=FMAX(d1,fabs(fc-gc));
        d1=FMAX(d1,fabs(fd-gd));
    }
    d2=0.0;
    for (j=1;j<=n2;j++) {                    其次，用第二个样本中的点作为原点
        quadct(x2[j],y2[j],x1,y1,n1,&fa,&fb,&fc,&fd);
        quadct(x2[j],y2[j],x2,y2,n2,&ga,&gb,&gc,&gd);
        d2=FMAX(d2,fabs(fa-ga));
        d2=FMAX(d2,fabs(fb-gb));
        d2=FMAX(d2,fabs(fc-gc));
        d2=FMAX(d2,fabs(fd-gd));
    }
    *d=0.5*(d1+d2);                          求K-S统计量的平均值
    sqen=sqrt(n1*n2/(double)(n1+n2));
    pearsn(x1,y1,n1,&r1,&dum,&dumm);         对每个样本求线性相关系数
    pearsn(x2,y2,n2,&r2,&dum,&dumm);
    rr=sqrt(1.0-0.5*(r1*r1+r2*r2));

    *prob=probks(*d*sqen/(1.0+rr*(0.25-0.75/sqen)));   用K-S概率函数 probks估算概率
}
```

参考文献和进一步读物：


Fasano, G. and Franceschini, A. 1987, *Monthly Notices of the Royal Astronomical Society*, vol. 225, pp. 155～170. [1]

Peacock, J.A. 1983, *Monthly Notices of the Royal Astronomical Society*, vol. 202, pp. 615～627 [2]

Spergel, D.N., Piran, T., Loeb, A.Goodman, J., and Bahcall, J.N. 1987, *Science*, vol. 237 pp. 1471～1473. [3]


## 14.8 萨维兹凯-戈雷平滑滤波器

在第13.5节我们已对数字滤波器的结构和应用有所了解，但对一些特殊用处的滤波器知之甚少。当然这取决于我们要用滤波器完成什么事情。对**低通滤波器**而言，其明显的用途是平滑噪声数据。

数据平滑的前提是，人们正在测量的变量是慢变化的并受到随机噪声的干扰。有时用某一点周围值的平均值来代替该点的值是非常有用的。这是由于周围点值与该点的值差别不大，平均只会减少噪声而不会给该点值带来太大的偏差。

我们必须指出，数据的平滑是针对稍区的，因而已超出了某些更值得提出和推荐的技术的范围，这些技术在本书其它部分讨论。例如，如果正在拟合数据为一个参数模型(见第15节)，利用原始数总是比利用已被预先平滑处理过的数据好。另一种使数据平滑相形见拙的技术被称为"最优化"或维纳滤波(这已在第13.3节中讨论，更详尽的见第13.3节)。数据平滑最适合于图形技术，能消除所有带有较大误差障碍的数据

点；或是一种能从图形中做出初步而又粗糙的简单参数估算的工具。

在这一节，我们将讨论一种特殊的低通滤波器，它适合于数据平滑，被称之为萨维兹凯-戈雷(Savitzky-Golay)[1]、最小二乘方[2]或DISPO(数据平滑多项式)[3]滤波器。不象通常那样先在频域中定义特性，然后转换到时间域，S-G 滤波器是直接来自时间域内数据平滑问题的一种特殊公式(正象现在我们将看到的)。S-G 滤波器起初(现在仍然经常地)用于提取可见光在噪声谱数据中谱线的宽度和高度。

让我们回顾一下，一个数字滤波器是应用于一列等间隔的数据序列 $f_i \equiv f(t_i)$，这里 $t_i = t_0 + i\Delta$，其中间隔 $\Delta$ 为常数，$i = \cdots -2, -1, 0, 1, 2, \cdots$ 从第13.5节我们看到最简单的数字滤波器(非递归或有限脉冲响应滤波器)是将每一个数据值 $f_i$ 用其本身和邻近的值 $g_i$ 的线性组合来代替：

$$g_i = \sum_{n=-n_L}^{n_R} c_n f_{i-n} \tag{14.8.1}$$

其中 $n_L$ 是所使用的 $i$ 左边的数据个数(在它之前)，$n_R$ 则是右边的(即在它之后)的数据个数。所之为**因果滤波器**的 $n_R$ 应等于零。

在讨论 S-G 滤波器时，我们首先考虑最简单的平均过程：对固定的 $n_L = n_R$，计算每一个 $g_i$(作为从 $f_{i-n_L}$ 到 $f_{i-n_R}$ 数据点的平均)，有时称它为**活动窗口平均**，它相当于 $c_n = 1/(n_L + n_R + 1)$ 的式(14.8.1)。如果基础函数是常数或随时间线性变化(增加或减少)，那么结果中不会引入任何偏差，在平均间隔一端的较高点被另一端的较低点所平衡。然而，如果基础函数有二阶非零导数，则偏差被引入。例如在一最极大值点处活动窗口平均总是减少了此处函数值。例如在光谱测定的应用中，使得一束窄谱线其高度减少、宽度增加。因为这些参量本身有物理意义，所以不希望引入偏差。

应注意，活动窗口平均保留了谱线下的区域(即零阶矩)，如果窗口是对称的，那么还保留了它在时间上的平均位置(即一阶矩)。受到扰动的是二阶矩，等价于线宽。

S-G 滤波的设计思想是能寻找到合适的滤波系数 $c_n$ 以保护高阶矩，也就是说，在对基础函数进行近似时，不是用常数窗口(其的估算是均值)，而是用高阶多项式(如二次、四次)：对每个点 $f_i$，我们利用最小二乘方拟合，对该窗口的所有 $n_L + n_R + 1$ 个点拟合成一个多项式，然后令 $g_i$ 为多项式在 $i$ 点的值(如果不熟悉最小二乘方拟合，可以先看一下第十五章)。我们并不使用在其它点处多项式的值。当窗口移动到下一点 $f_{i+1}$ 时，我们将对新窗口进行新的最小二乘方拟合。

如果照此做下去，显然所有最小二乘方拟合是相当繁复的。幸运的是，由于最小二乘方拟合过程仅涉及一个线性矩阵取反，则拟合多项式系数本身与数据值成线性关系数，这意味着我们可以事先对一组假设值(除一个1外其它都取零)做拟合，然后通过线性组合来拟合真实数据。这里的关键是：对式(14.8.1)而言，存在一组特殊的滤波系数 $c_n$，能"自动"完成在活动窗口内的最小二乘方拟合过程。

为了导出这样的系数，让我们考虑 $g_0$ 是怎样得到的。我们希望在第 $i$ 窗口中对值 $f_{-n_L}, \ldots, f_{n_R}$ 拟合一个 $M$ 阶多项式(即 $a_0 + a_1 i + \ldots + a_M i^M$)，那么 $g_0$ 即为 $i=0$ 时多项式的值，即 $g_0 = a_0$。对这一问题的设计矩阵为(第15.4节)：

$$A_{ij} = i^j \qquad i = -n_L, \ldots, n_R, \qquad j = 0, \ldots, M \tag{14.8.2}$$

向量 $\{a_j\}$ 和 $\{f_i\}$ 的正规的方程以矩阵形式给出：

$$(\mathbf{A}^T \cdot \mathbf{A}) \cdot \mathbf{a} = \mathbf{A}^T \cdot \mathbf{f} \quad \text{或} \quad \mathbf{a} = (\mathbf{A}^T \cdot \mathbf{A})^{-1} \cdot (\mathbf{A}^T \cdot \mathbf{f}) \tag{14.8.3}$$

我们也可得到：

$$\{\mathbf{A}^T \cdot \mathbf{A}\}_{ij} = \sum_{k=-n_L}^{n_R} A_{ki} A_{kj} = \sum_{k=-n_L}^{n_R} k^{i+j} \tag{14.8.4}$$

和

$$\{\mathbf{A}^T \cdot \mathbf{f}\}_j = \sum_{k=-n_L}^{n_R} A_{kj} f_k = \sum_{k=-n_L}^{n_R} k^j f_k \tag{14.8.5}$$

因为系数 $c_n$ 是当 f 用单位向量 $e_n$（$-n_L<n<n_R$）代换时的 $a_0$，则有：

$$c_n = \{(\mathbf{A}^T\cdot\mathbf{A})^{-1}\cdot(\mathbf{A}^T\cdot\mathbf{e}_n)\}_0 = \sum_{m=0}^{M}\{(\mathbf{A}^T\cdot\mathbf{A})^{-1}\}_{0m}n^m \tag{14.8.6}$$

注意，式(14.8.6)表明我们仅需一行逆矩的值(数值计算上讲，仅用单一的后代换的 LU 分解就可以得到这一值)。

下面的 **savgol** 是将式(14.8.6)程序化，作为输入，它取参数 nl$=n_L$、nr$=n_R$ 和 m$=M$，输出数组 c 的物理长度 np 也是输入量，数据拟合的参量 ld 将是零。实际上，ld 指定了 $\{a_i\}$ 中哪一个系数将返回，这里我们仅对 $a_0$ 感兴趣。为了另外的目的，即计算数值微分(第5.7节中已提及)，则有用的选择是 ld≥1，例如 ld=1 时滤出的一阶导数是式(14.8.1)的卷积除变化量 Δ。对求导而言，人们总是希望 $m=4$ 或更大些。

```
#include <math.h>
#include "nrutil.h"

void savgol(float c[], int np, int nl, int nr, int ld, int m)
    在c[1..np]中返回一组S-G滤波系数，它是以环绕次序排列，与程序convlv中的变量respns是一致的，nl是所使
    用的左边(过去)数据点的个数，而nr是右边(将来)数据点的个数，而所使用的全部数据点个数为nl+nr+1. ld是
    所求导数的阶(即，对平滑函数ld=0)，m是平滑多项式的阶，即等于最高的保留矩：通常m=2或m=4。
{
    void lubksb(float **a, int n, int *indx, float b[]);
    void ludcmp(float **a, int n, int *indx, float *d);
    int imj,ipj,j,k,kk,mm,*indx;
    float d,fac,sum,**a,*b;

    if (np < nl+nr+1 || nl < 0 || nr < 0 || ld > m || nl+nr < m)
    nrerror("bad args in savgol");
    indx=ivector(1,m+1);
    a=matrix(1,m+1,1,m+1);
    b=vector(1,m+1);
    for (ipj=0;ipj<=(m << 1);ipj++) {       设置所求最小二乘方拟合的正规方程
        sum=(ipj ? 0.0 : 1.0);
        for (k=1;k<=nr;k++) sum += pow((double)k,(double)ipj);
        for (k=1;k<=nl;k++) sum += pow((double)-k,(double)ipj);
        mm=FMIN(ipj,2*m-ipj);
        for (imj = -mm;imj<=mm;imj+=2) a[1+(ipj+imj)/2][1+(ipj-imj)/2]=sum;
    }
    ludcmp(a,m+1,indx,&d);                  求解：LU分解法
    for (j=1;j<=m+1;j++) b[j]=0.0;
    b[ld+1]=1.0; 右端向量是单位向量，依赖于所求导数

    lubksb(a,m+1,indx,b);                   求逆矩阵之行
    for (kk=1;kk<=np;kk++) c[kk]=0.0;       零化输出矩阵(可能大于系数的个数)
    for (k = -nl;k<=nr;k++) {
        sum=b[1];                           每个S-G系数是整数幂与逆矩阵行的点乘
        fac=1.0;
        for (mm=1;mm<=m;mm++) sum += b[mm+1]*(fac *= k);
        kk=((np-k) % np)+1;                 存贮环绕的顺序
        c[kk]=sum;
    }
    free_vector(b,1,m+1);
    free_matrix(a,1,m+1,1,m+1);
    free_ivector(indx,1,m+1);
}
```

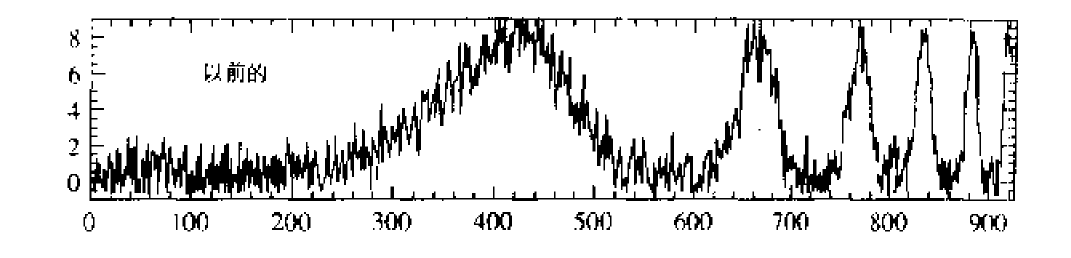
作为输出，savgol 给出了系数 $c_n$（$-n_L\leqslant n\leqslant n_R$），这些都以"环绕顺序"的形式储存在 c 中，即 $c_0$ 在 c[1]，$c_{-1}$ 在 c[2]中，(对进一步的负标亦类推)等等；$c_1$ 则存储在 c[np]，$c_2$ 在 c[np－1](对更大的正括号亦类推)等。这一顺序看起来挺神密，但对因果滤波器来说却很自然，它在低的 c 数组元素内有非零系数，这也是第13.1节中函数 convlv 所要求的顺序，它是对一数据组进行数字滤波时所采用的次序。

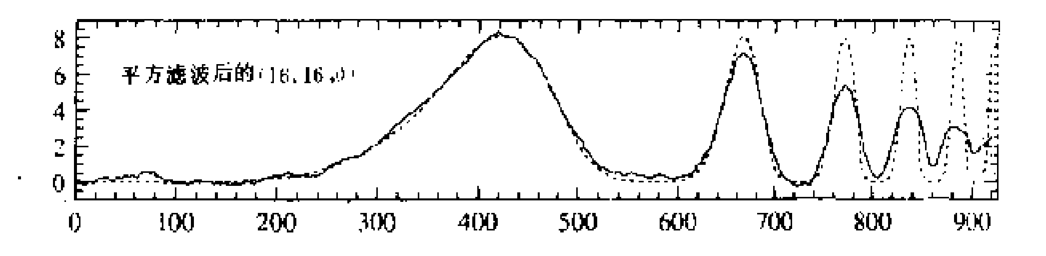
表14.8.1中给出了 savgol 的某些标准输出。对于 2 阶和 4 阶的情况，给出几种 $n_L$ 和 $n_R$ 选择的 S-G 滤

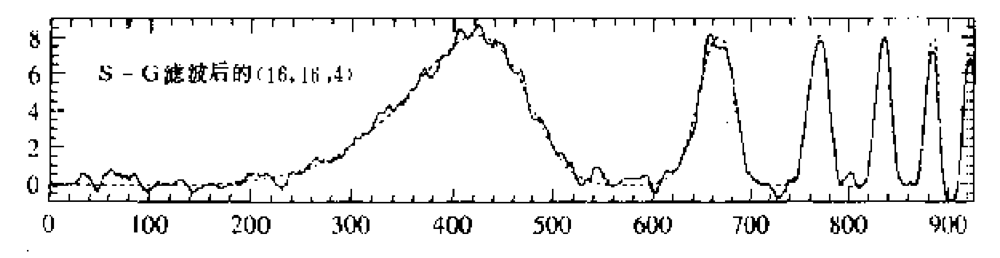
波器系数;中间一列是为求得被平滑的 $g_i$ 而应用于数据 $f_i$ 的系数;左边的系数用于以前的数据;右边用于以后的数据。系数之和总是 1 (没有舍入误差)。人们看到,作为平滑算子,系数总是在中心有正的凸起,但常有小的、远离中心的成对的正的和负的校正。事实上,S-G 滤波器最适合于非常大的 $n_L$ 和 $n_R$,因为这些点的公式可用相对少量的平滑完成。

**表14.8.1**

| $M$ | $n_L$ | $n_R$ | 样本 Savitzky-Golag 系数 | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 2 | 2 | | | | -0.086 | 0.343 | 0.486 | 0.343 | -0.086 | | | |
| 2 | 3 | 1 | | | -0.143 | 0.171 | 0.343 | 0.371 | 0.257 | | | | |
| 2 | 4 | 0 | | 0.086 | -0.143 | -0.086 | 0.257 | 0.886 | | | | | |
| 2 | 5 | 5 | -0.084 | 0.021 | 0.103 | 0.161 | 0.196 | 0.207 | 0.196 | 0.161 | 0.103 | 0.021 | -0.084 |
| 4 | 4 | 4 | | 0.035 | -0.128 | 0.070 | 0.315 | 0.417 | 0.315 | 0.070 | -0.128 | 0.055 | |
| 4 | 5 | 5 | 0.042 | -0.105 | -0.023 | 0.140 | 0.280 | 0.333 | 0.280 | 0.140 | -0.023 | -0.105 | 0.042 |




顶端:合成的被噪声干扰的数据,它由一系列不断变窄的隆起部分和附加的高斯白噪声组成。中间:用简单的活动窗平均进行数据平滑的结果。这个窗左边延伸16个点,右边延伸16个点,总共33个点。注意,其较窄隆起部分变宽而且相应幅度变低。点划的曲线是表示原产生合成数据的基本函数。底端:用 S-G 平滑滤波器进行数据平滑的结果(阶为4),采用了同样的33个点。其对较宽的隆起部分的平滑性稍差,但较窄的隆起部分的高度和宽度得到了保护。

图14.8.1

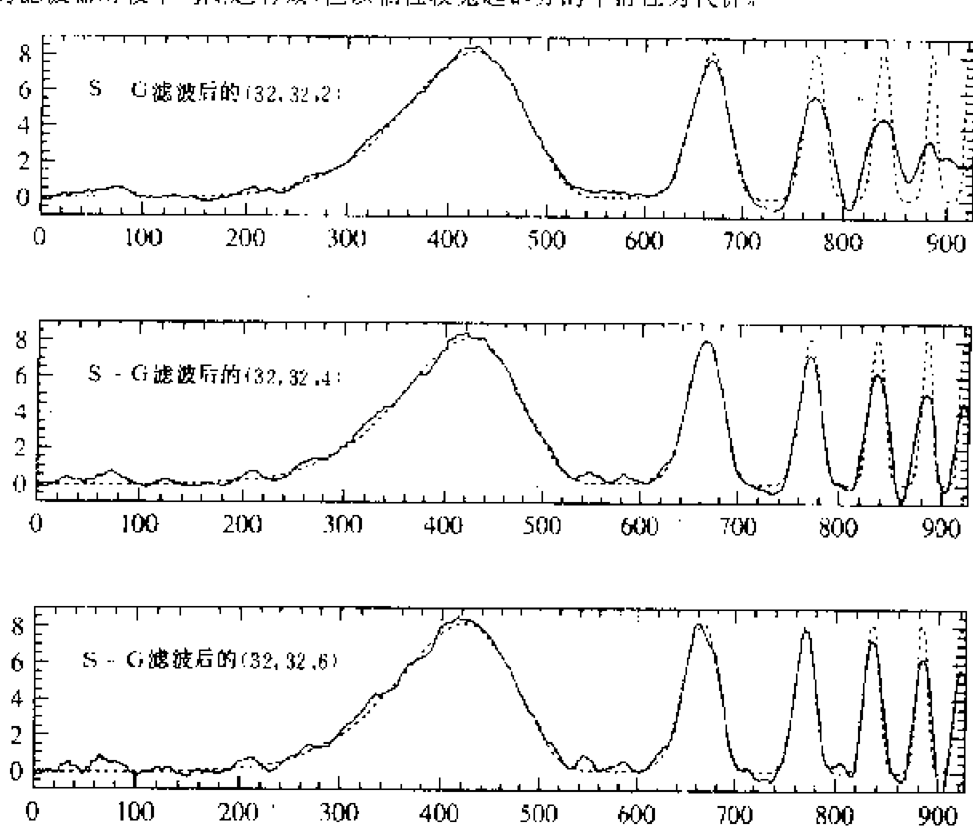
图14.8.1给出了使用 33 个点平滑滤波器的数值实验,即 $n_L=n_R=16$ 。最上一图为被测试的函数,有六

个不同宽度(都是 8 个单位高的)隆起处。对于这个函数，单位方差的高斯白噪声已经被叠加上了(没有被噪声干扰的测试函数由中间和下面图中的点划线给出)，隆起部分的宽度(最大值一半处的全宽度或 FWHM)分别是 140、43、24、17、13 和 10 。

图14.8.1的中间部分给出了用活动窗平均平滑的结果，可以看出，宽度为 33 的窗口在平滑最宽的隆起部分非常有效，而对较窄的隆起则存在相当大的高度降低和宽度增加。用它描述原基本函数是非常差的。

最下面图给出了同样宽度，$M=4$ 的 S-G 滤波器平滑的结果，可以看出隆起部分的高度和宽度都不寻常地保留住了，唯一的缺陷是最宽的隆起部分欠平滑。这是因为 S-G 滤波器系数的主要的正的凸处仅是整个 33 点宽度的分数。可以猜想最好的结果应在，4 阶 S-G 滤波器的全宽度是所期望的数据的 FWGM 1 至 2 倍时得到(参阅[3]和[4])。

图14.8.2 给出用较宽的三个不同的阶滤波器平滑相同"噪声"的结果，其中 $n_L=n_R=32$ ( 65 个点的滤波器)，$M=2$、4、6。可以看出，当隆起部分相对于滤波器的大小太窄时，则 S-G 滤波器也会在某些点发散，较高阶的滤波器对较窄的隆起有效，但以牺牲较宽起部分的平滑性为代价。




顶图：阶为 2，中间：阶为 4，底图：阶为 6。对于窄部分的分辨率来说，所有这些滤波器都是令人不满意的，高阶滤波器能较好地保留高度到宽度，但对较宽部分的平滑较差。

图14.8.2 对与图14.8.1相同的数据组，使用较宽的 65 点 S-G 滤波器的结果

总而言之，在限制条件下，S-G 滤波器能提供没有分辨率损失的平滑，做到这点是基于假设相对稀疏的数据点具有能用来减少噪声电平的有效的冗余度。所设冗余度的特点是基础函数应该被多项式拟合。如果这点成立，当平滑线轮廓相对于滤波器宽度不是太窄时，S-G 滤波器的效果相当惊人。如果这点不成立，这一滤波器与其它滤波器相比就没有太大优势。

最后要讨论的是不规则取样数据，即 $f_i$ 在时间轴上不是均匀分隔的。这种情况下，推广 S-G 滤波必须对每个数据点的活动窗口做最小二乘方拟合，每个左右都包括固定的数据点 $n_L$ 和 $n_R$。因为是不规则分隔，

没有任何办法得到用于一个点以上的一般性的滤波系数，人们必须对每个点都做最小二乘拟合，当 $n_L$、$n_R$ 和 $M$ 较大时，将会遇到特别繁重的计算。

作为一简单的替代方法，即仍把数据点视为等间隔的（实际上不是），这相当于，在每个滤波窗口中，移动数据点到等间隔位置。这样的位移会给函数引入新的等效附加噪声源。在这种情况下，平滑是有用的，因为这种噪声远小于实际存在的噪声。特别是，如果在窗内点的位置近似随机的，则一个粗糙的判据为：如果在整个宽度 $N=n_L+n_R+1$ 点的窗口内，$f$ 的变化小于 $\sqrt{N/2}$ 倍的单一点的噪声测量值，则这一简单方法可以使用。

## 参考文献和进一步读物：


Savitzky A., and Golay, M. J. E. 1964, *Analytical Chemistry*, vol. 36, pp. 1627~1639. [1]

Hamming, R. W. 1983, *Digital Filters*, 2nd ed. (Englewood Cliffs, NJ: Prentice-Hall). [2]

Ziegler, H. 1981, *Applied Spectroscopy*, vol, 35, pp. 88~92. [3]

Bromba. M. U. A., and Ziegler, H. 1981, *Analytical Chemistry*, vol. 53, pp. 1583~1586. [4]

# 第十五章　数据的模型建立

## 15.0　引言

给定一组观察结果，我们经常要对这组数据进行浓缩和概括，使它适合参数可调整的模型。有时这种模型是一些简单方便的函数，例如多项式或高斯函数，从而让拟合的结果提供正确的系数。有时，模型的参数来自于数据所满足的基本理论，例如复杂的化学反应系统中的变化率方程的系数，或者一个双星的轨道参数。模型化还可以用作于一种约束的插值方法，从中可以延伸出一些数据，形成一个连续函数。但是，对于这些函数的大致形状应有一个基本的概念。

对各种情况，基本方法通常是一样的：选择或设计一个**优值图形函数**（简称为**优值函数**），它必须能度量数据和具有特别选择参数的模型之间一致性的程度。接着，对模型的参数进行调节，使优值函数取最小值，产生**最佳拟合参数**。这种调节过程也就是一个多维求极小化的问题。这种优化问题是第十章中的课题；但是，还有一种特殊的、更有效的方法，具体地说就是模型建立的方法，我们将在这一章中讨论。

除了寻找最佳拟合参数外，还有一些重要的问题。数据通常是不准确的，它们存在**测量误差**（在信号处理的文献中称为**噪声**）。因为，典型的数据永远不会精确地拟合于正在使用的模型，即便这种模型是正确的。因此，我们要有估价模型是否恰当的方法，也就是说要有检验**拟合优度**的某些统计标准。

我们通常还必须知道，由数据集确定的参数的准确程度。换言之，我们要知道最佳拟合参数的似然误差。

最后，我们通常发现优值函数不是单峰的，不具单一的极小值。在有些情况下，我们可能感兴趣的是全局而不是局部的问题，不是"这种拟合优度如何？"的问题，而是"在参数空间的某个区域，如何确信没有比这种拟合更好的了？"的问题。正如第十章所述，特别是第10.9节，这类问题通常很难解决。

我们要提请注意的是，参数拟合不是一个参数估计的终结过程。一个真正有用的拟合过程必须提供：(i)拟合的参数；(ii)拟合参数的误差估计；以及(iii)拟合优度的统计度量。如果(iii)中的结果表明这种模型不和数据一致，那么(i)和(ii)中的结果也就几乎没有价值。遗憾的是，许多参数估计的实践者永远只进行到第(i)步！如果数据的图表和模型"看起来很好"，他们就认为拟合是可接受的。这种方法被称为"肉眼的 $\chi$ 方法"。庆幸的是，这种方法的实践者得到了他们应得的惩罚。

## 15.1　最大似然估计的最小二乘方法

假定我们用 $M$ 个可调参数 $a_j(j=1,...,M)$ 的模型来拟合 $N$ 个数据点 $(x_i,y_i)$，$i=$

$1,\ldots,N$。这种模型给出已测变量和因变量之间的一个函数关系

$$y(x)=y(x;a\ldots a_M) \tag{15.1.1}$$

上式右边明显地标出了因变量和参量间的依赖关系。

为了得到 $a_j$ 的拟合值,我们要求对什么求极小化呢?我们首先想到的是,熟悉的最小二乘方拟合,通过 $a_1,\ldots,a_M$ 求

$$\sum_{i=1}^{N}[y_i-y(x_i;a\ldots a_N)]^2 \tag{15.1.2}$$

的极小值。但这个式子是从哪里来的呢?它以什么基本原理为基础呢?为了回答这个问题,我们必须进行**最大似然估计**主题的讨论。

给定一组特定的数据 $x_i$ 和 $y_i$,我们有一种直觉,某几组参数值 $a_1\cdots\cdots a_M$ 是非常不可能的——对这些参数,模型函数 $y(x)$ 和数据相差甚远;但是另一些参数却非常可能——对于这些参数,模型函数 $y(x)$ 和数据非常相似。可是我们怎样将这种直觉作数量化呢?我们怎样选择看起来是正确的拟合参数呢?若问"一组拟合的参数 $a_1\cdots\cdots a_M$ 正确的可能性有多大?"这样的问题是毫无意义的,原因是不存在模型的统计全域,从中可以导出参数,而只是一个模型,一个正确模型,和一组数据的统计全域,从中可导出数据集。

实际情况尽管如上所述,但是我们可以将问题转换一下,问"**给定一组特定的参数**,这组数据集发生的概率多大?"在这样的问题中,如果 $y_i$ 取连续的值,概率永远是零,除非我们加上这样的语句,"……对于每个数据点上或减去某个固定的 $\Delta y$"。所以让我们永远将这句话看做是不讲自明的。如果得到的数据组的概率无限小,则我们可以认为被考虑的参数是不正确的。同样,直觉告诉我们,对于正确选择的参数,数据集也不会不合适。

换言之,我们用给定参数下数据的概率(在数学上它是可计算的数)作为给定数据下的参数的似然。这种借鉴完全基于直觉,它没有任何正式的数学机理。正如我们已经指出的一样,统计不是数学的一个分支。

一旦我们有了这种直觉的借鉴,那么我们只要前进小小的一步,就能精确地拟合出参数 $a_1\cdots\cdots a_M$,也就是通过求上面定义的似然的最大值来确定这些参数。这种参数估计方法就是**最大似然估计**。

我们现在要和式(15.1.2)联系起来。假设每个数据点 $y_i$ 的测量误差是随机独立的,并且围绕"真实"模型 $y(x)$ 呈正态(高斯)分布。还假设这些正态分布的标准差 $\sigma$ 对所有的数据点都是一样的。那么,数据集的概率为各个数据点概率的乘积,

$$P\propto\prod_{i=1}^{N}\left\{\exp\left[-\frac{1}{2}\left(\frac{y_i-y(x_i)}{\sigma}\right)^2\right]\Delta y\right\} \tag{15.1.3}$$

注意,在乘积的每一项中有因子 $\Delta y$。求式(15.1.3)的极大值等价于极大化它的对数,或者极小化它的负对数,即

$$\left[\sum_{i=1}^{N}\frac{[y_i-y(x_i)]^2}{2\sigma^2}\right]-N\log\Delta y \tag{15.1.4}$$

因为 $N$、$\sigma$ 和 $\Delta y$ 都是常数,求上式的极小值等价于求式(15.1.2)的极小值。

正如我们上面讨论的,如果测量误差是独立的,并且服从具有常数方差的正态分布,则最小二乘方拟合就是拟合参数的最大似然估计。注意,我们对模型 $y(x;a\cdots)$ 中的参数 $a_1\cdots a_M$ 是线性的还是非线性的没做任何假设。在下面,我们将不假设方差为常数的而得到非常

相似的式子，它被称为"$\chi^2$拟合"或"权重最小二乘方拟合"。但是，首先我们还是进一步讨论严格假设下的正态分布。

在一百年左右的时间里，数理统计学家们对于这样一个事实——大量非常小的随机偏差之和的概率分布收敛于一个正态分布(对于更准确的是**中心极限定理**的叙述，查阅 V·米斯(Von Mises)或其它的数理统计的一般著作)——一直满怀激情。这种激情使人们的注意力远离了这样一个事实，即对实际的数据来说，如果能实现的话，正态分布常常是非常粗糙的实现。我们经常而不是偶然地被告知，平均地看，测量值在真值的$\pm\sigma$范围内的概率是95%，在$\pm3\sigma$范围内的概率是99.7%，继续扩展下去，我们能预料测量的结果在$\pm20\sigma$以外的概率为$1/2\times10^{88}$。我们都知道"假信号"也比这种情况更为可能发生！

在有些情况中，正态分布的偏差很容易理解和数量化。例如在计数事件的测量中，测量的误差通常表现为泊松(Poisson)分布，它的积累概率函数已经在第6.2节中讨论过。当涉及数据点的计数数目非常大时，泊松分布趋向于高斯分布，但是当按相对精度测量时，这种收敛不是一致的。分布尾部的标准差越多，则使某个数值实现趋于高斯分布所需的计数数目越大。这种效应的表现永远是一样的：高斯分布预计的"尾部"事件远不像它们的实际情况(由泊松分布确定)。这就导致这样一些事件，如果发生的话，它比实际情况更容易偏离最小二乘方拟合。

在另外一些情况下，一个正态分布的偏差在一些细节上难于掌握。有些实验数据点有时**"偏离正道"**。也许在某一点的测量时，电源突然有一点起伏，或者某人踢了仪器一脚，或者某人写下了一个错误的数字。这种点被称为**"偏离点"**。它们非常容易把原本对其它数据十分适合的最小二乘方拟合变得毫无意义。在高斯模型的假设下，它们发生的概率如此小，使得最大似然估计错误地畸变曲线，而把它们处于直线状态。

**稳健统计**是解决当正态即高斯模型是一个比较差的逼近，或者"偏离点"非常重要情况下的课题。我们将在第15.7节中简洁地讨论稳健统计方法。从现在起一直到第15.7节我们都假设测量误差符合高斯模型。非常重要的一点是，必须记住这个模型的局限性，特别在应用由此假设中导出来的一些有用的方法时更应如此。

最后要注意的是，我们对测量误差的讨论局限于**统计误差**，这种误差在取大量的数据进行平均时是能去掉的。测量还易产生**系统误差**，它是不能通过大量数据平均消去的。例如，金属米棒的校准依赖于温度。如果我们的测量在同一错误的温度下进行，则任何数目的平均或数据处理都不能校正这不可知的系统误差。

### 15.1.1 $\chi^2$拟合

在前面即第14.3节我们曾经讨论过$\chi^2$统计，这里将在稍微不同的情况下进行讨论。

如果每个数据点$(x_i, y_i)$有它自己的标准差$\sigma_i$，那么只要把方程(15.1.3)略微变化一下，在符号$\sigma$下加一个下标$i$。这个下标同样自然地加进式(15.1.4)，以致模型参数的最大似然估计成为求下面数量的最小值：

$$\chi^2 \equiv \sum_{i=1}^{N}\left(\frac{y_i - y(x_i; a_1 \ldots a_M)}{\sigma_i}\right)^2 \tag{15.1.5}$$

这个量被称为"chi 平方"。

测量误差不论是何种程度的正态分布，量$\chi^2$均是相应的$N$个正态分布量的平方之和，

其中每个量均归一化成单位方差。一旦我们调整参数 $a_1 \cdots\cdots a_M$ 使得 $\chi^2$ 取极小值，则在和式中的各项就不完全是统计独立的。但是对于参数是线性的模型来说，已经说明在最小值处，不同取值的 $\chi^2$ 的概率分布还是能解析地导出的，它符合**自由度为 N－M 的** chi **平方分布**。我们在第6.2节已经知道怎样用不完全的伽马(gamma)函数 **gammq** 来计算这种概率函数。特别是方程(6.2.18)给出了 chi 平方偶然超过某一特定的 $\chi^2$ 值的概率 $Q$，其中 $\nu = N - M$ 是**自由度个数**。量 $Q$，或者它的互补量 $P = 1 - Q$ 经常在统计书本中以列表形式给出，但是我们发现用 **gammq** 非常简单，例如计算我们要的值 $q$=**gammq**$(0.5*\nu, 0.5*\chi^2)$。对于参数 $a$ 不是严格的线性模型时，假设 $\chi^2$ 分布仍然是适用的。这种做法很普遍，不会有大的错误。

这个计算出来的概率值可以做为模型拟合好坏的标准。如果对于某一数据组，$Q$ 是一个非常小的概率数值，那么这种明显的不一致，不可能是偶然的波动导致的。更可能是(i)模型是错误的——从统计的观点应被抛弃，或者(ii)某人错误地告诉了测量误差 $\sigma_i$ 的大小——它们实际上比所陈述的要大。

还有很重要的一点，那就是 chi 平方的概率 $Q$ 并不能衡量测量误差呈正态分布的假设的可信度。我们只是假设误差确实如此，在多数但非全部情况下，非正态分布误差的效果是产生许多"偏离点"。这些点使概率值 $Q$ 减小，所以除了上面列出的两点外，我们还可以增加一个可能的但不是确定的结论：(iii)测量误差可能不是呈正态分布。

(iii)的发生是很普遍的，并且也相当温和。正是由于这种原因使得明智的实验者能容忍低概率值 $Q$。对于任何模型我们通常还可以确定一个可接受的 $Q$ 值，比如 $Q > 0.001$。这并不象它看起来那样粗糙：真正的错误模型会由于非常小的 $Q$ 值比如 $10^{-18}$ 而被舍弃。但是如果天天接受 $Q \sim 10^{-3}$ 的模型，就应该找找原因。

如果碰巧知道测量误差的实际分布规律，那么就可以用**蒙特卡罗**(Monte Carlo)**模拟**的方法从一个特定的模型中得出数据，如第7.2～第7.3节所述。接着可以将模拟数据应用于实际拟合过程，不但可以确定 $\chi^2$ 统计的概率分布，还可以确定由拟合所得出的模型参数精确性。我们将在第15.6节对此作进一步的讨论。它的技巧是非常一般的，但它的代价也非常昂贵。

在另外一个相反的极端，有些时候概率值 $Q$ 太大，接近于 1，简直好得以致于不可能是真的了！非正态的测量误差一般不可能导致这种弊端，因为正态分布已达到任何一个可能的分布所能达到的"紧凑"。一个太好的 chi 平方拟合的原因几乎永远是，实验者在一种保守的拟合中过高估计了他或她的测量误差。还有一种非常小的可能，一个太好的 chi 平方拟合实际上标志着一种欺骗，为适合模型而捏造出数据。

首要的规则是，一个拟合"中等"好的典型 $\chi^2$ 值是 $\chi^2 \approx \nu$。更准确的表述是，随着 $\nu$ 的增大，统计量 $\chi^2$ 趋近于正态分布，均值为 $\nu$，标准差为 $\sqrt{2\nu}$。

在某些情况下，和一组测量数据相联系的不确定性是不能事先知道的，并且考虑用有关 $\chi^2$ 的拟合要导出 $\sigma$ 的值。如果我们假设所有的测量有相等的标准差，$\sigma_i = \sigma$，模型拟合得也很好，那么我们可以采取下面的步骤。首先对所有的点任意假设一个常数 $\sigma$，接着通过求 $\chi^2$ 的极小值拟合模型参数，最后重新计算

$$\sigma^2 = \sum_{i=1}^{N} [y_i - y(x_i)]^2/(N - M) \tag{15.1.6}$$

显然，这种方法使我们不能对拟合优度进行独立的估计，这也是被它的实践者偶然忽略的问

题。但是，在测量误差不知道的情况下，这种方法至少能为各个数据点提供某种误差上限。

如果我们在式(15.1.5)中分别对参数 $a_k$ 求微分，我们就得到使 chi 平方取最小值的方程组：

$$0=\sum_{i=1}^{N}\left(\frac{y_i-y(x_i)}{\sigma_i^2}\right)\left(\frac{\partial y(x_i;\ldots a_k\ldots)}{\partial a_k}\right)\qquad k=1,\ldots,M \tag{15.1.7}$$

一般情况下，方程(15.1.7)是含有 $M$ 个未知参数 $a_k$ 的 $M$ 个非线性方程组。在本章随后所叙述的许多过程都由式(15.1.7)及它的特例导出。

## 15.2 拟合数据成直线

一个具体的实例将使上节的讨论更有意义。我们考虑一个用 $N$ 个数据点拟合成直线的问题，直线模型为

$$y(x)=y(x;a,b)=a+bx \tag{15.2.1}$$

这个问题被称为**线性回归**，这个术语很久以前源于社会科学。我们假设和每个测量值 $y_i$ 相联系的不确定性值 $\sigma_i$ 是知道的，还假设 $x_i$ 的值(自变量)是精确知道的。

为了衡量模型和数据的一致程度，我们采用 chi 平方优值函数(15.1.5)，在直线拟合这种情况下，式(15.1.5)变为

$$\chi^2(a,b)=\sum_{i=1}^{N}\left(\frac{y_i-a-bx_i}{\sigma_i}\right)^2 \tag{15.2.2}$$

如果测量误差是正态分布，则上面的优值函数将给出参数 $a$ 和 $b$ 的最大似然估计；如果测量误差不是正态分布，则这种估计将不是最大似然估计，但在实际应用中仍然有用的。在第15.7节我们将处理偏离点太多以致优值函数 $\chi^2$毫无用处的情况。

求方程(15.2.2)的最小值来确定 $a$ 和 $b$。将 $\chi^2(a,b)$分别对 $a$ 和 $b$ 求导，并令其等于0：

$$\begin{aligned}0&=\frac{\partial\chi^2}{\partial a}=-2\sum_{i=1}^{N}\frac{y_i-a-bx_i}{\sigma_i^2}\\0&=\frac{\partial\chi^2}{\partial b}=-2\sum_{i=1}^{N}\frac{x_i(y_i-a-bx_i)}{\sigma_i^2}\end{aligned} \tag{15.2.3}$$

如果我们定义下面的符号，则上式可以再写成一个比较简便的形式：

$$\begin{aligned}&S\equiv\sum_{i=1}^{N}\frac{1}{\sigma_i^2}\qquad S_x\equiv\sum_{i=1}^{N}\frac{x_i}{\sigma_i^2}\qquad S_y\equiv\sum_{i=1}^{N}\frac{y_i}{\sigma_i^2}\\&S_{xx}\equiv\sum_{i=1}^{N}\frac{x_i^2}{\sigma_i^2}\qquad S_{xy}\equiv\sum_{i=1}^{N}\frac{x_iy_i}{\sigma_i^2}\end{aligned} \tag{15.2.4}$$

有了这些定义，式(15.2.3)变为

$$\begin{aligned}aS+bS_x&=S_y\\aS_x+bS_{xx}&=S_{xy}\end{aligned} \tag{15.2.5}$$

这两个方程中的两个未知数可计算如下：

$$\Delta\equiv SS_{xx}-(S_x)^2$$

$$a=\frac{S_{xx}S_y-S_xS_{xy}}{\Delta}$$

$$b = \frac{SS_{xy} - S_x S_y}{\Delta} \tag{15.2.6}$$

方程(15.2.6)给出了最佳模型拟合参数的答案。

但是我们不能到此结束。因为数据的测量误差必然会导致拟合参数不确定度，所以我们必须估计 $a$ 和 $b$ 的不确定度。如果各个数据点是独立的，则每个数据点的不确定度都会在参数的不确定度中产生相应的部分。根据误差传播的考虑，我们得到任何函数值的方差是

$$\sigma_f^2 = \sum_{i=1}^{N} \sigma_i^2 \left(\frac{\partial f}{\partial y_i}\right)^2 \tag{15.2.7}$$

对于上面的直线情形，将式(15.2.6)中的 $a$ 和 $b$ 分别对 $y_i$ 求导，得到

$$\begin{aligned} \frac{\partial a}{\partial y_i} &= \frac{S_{xx} - S_x x_i}{\sigma_i^2 \Delta} \\ \frac{\partial b}{\partial y_i} &= \frac{S x_i - S_x}{\sigma_i^2 \Delta} \end{aligned} \tag{15.2.8}$$

然后，按照式(15.2.7)那样对各个数据点求和，我们得到

$$\begin{aligned} \sigma_a^2 &= S_{xx}/\Delta \\ \sigma_b^2 &= S/\Delta \end{aligned} \tag{15.2.9}$$

它们分别是 $a$ 和 $b$ 方差的估计。在第15.6节我们将看到，为了正确地刻划参数估计的可能不确定度，还需要增加一个量。这个量就是 $a$ 和 $b$ 的**协方差**，并且由下式给出：

$$\text{Cov}(a,b) = -S_x/\Delta \tag{15.2.10}$$

$a$ 和 $b$ 不确定度之间的相关系数是一个在$-1$ 和 1 之间的数值，根据式(15.2.10)(和等式14.5.1比较)可得

$$r_{ab} = \frac{-S_x}{\sqrt{SS_{xx}}} \tag{15.2.11}$$

正的 $r_{ab}$值说明 $a$ 和 $b$ 的误差可能同号，而负的 $r_{ab}$值说明 $a$ 和 $b$ 的误差是反相关，可能有相反的符号。

我们的工作还不能到此结束。我们还必须估计数据和模型之间的拟合优度。缺少这一估计，将使模型中的参数 $a$ 和 $b$ 显示不出任何意义！一个和式(15.2.2)所确定的值一样粗劣的 $\chi^2$值之概率 $Q$，由下式计算：

$$Q = \textbf{gammq}\left(\frac{N-2}{2}, \frac{\chi^2}{2}\right) \tag{15.2.12}$$

这里 **gammq** 是第6.2节中计算不完全伽马函数 $Q(a,x)$的程序。如果比 0.1 大，那么我们确信拟合得比较好，如果 $Q$ 比 0.001 大，只要误差是非正态分布或者适当地低估了一点，那么我们认为拟合是可以接受的。如果 $Q$ 比 0.001 小，那么模型或者估计过程有问题。对于后一种情况，请翻到第15.7节进行深入的讨论。

如果不知道每个数据点的测量误差 $\sigma_i$，而贸然用等式(15.1.6)估计这些误差，则可以采取如下的步骤估计参数 $a$ 和 $b$ 的可能不确定度：在式(15.2.6)前后所有的等式中设 $\sigma_i \equiv 1$，然后将式(15.2.9)中得到的 $\sigma_a$ 和 $\sigma_b$ 乘以附加的因子 $\sqrt{\chi^2/(N-2)}$，其中 $\chi^2$由拟合出来的参数 $a$ 和 $b$ 根据式(15.2.2)计算。和上面讨论的一样，这个过程等价于假设拟合得很好，因此得不到独立的衡量拟合优度的概率 $Q$。

在第14.5节中，我们保留了线性相关系数 $r$(式(14.5.1)确定)和拟合优度测量值 $\chi^2$(式(15.2.2)确定)之间的关系，对于无权重数据点(即所有的 $\sigma_i=1$)，这个关系是

$$\chi^2 = (1 - r^2) N \mathrm{Var}(y_1 \ldots y_N) \tag{15.2.13}$$

其中

$$N \mathrm{Var}(y_1, \ldots y_N) \equiv \sum_{i=1}^{N} (y_i - \bar{y})^2 \tag{15.2.14}$$

对于不同权重值的数据点，如果将式(14.5.1)中的求和乘以权重 $1/\sigma_i^2$，上式仍然有效。

下面的函数 **fit** 准确地实现了我们上面所讨论的运算。当权重 $\sigma$ 事先知道时，函数的计算和上面的公式完全一致。但是，当权重 $\sigma$ 不知道时，程序过程假设每个点有相同的 $\sigma$，而且还假设拟合得很好，这正是第15.1节中讨论的那样。

公式(15.2.6)对舍入误差很敏感，因此我们将它们重写如下：定义

$$t_i = \frac{1}{\sigma_i}\left(x_i - \frac{S_x}{S}\right), \quad i = 1, 2, \ldots, N \tag{15.2.15}$$

和

$$S_{tt} = \sum_{i=1}^{N} t_i^2 \tag{15.2.16}$$

那么，通过直接的代换可证明

$$b = \frac{1}{S_{tt}} \sum_{i=1}^{N} \frac{t_i y_i}{\sigma_i} \tag{15.2.17}$$

$$a = \frac{S_y - S_x b}{S} \tag{15.2.18}$$

$$\sigma_a^2 = \frac{1}{S}\left(1 + \frac{S_x^2}{S S_{tt}}\right) \tag{15.2.19}$$

$$\sigma_b^2 = \frac{1}{S_{tt}} \tag{15.2.20}$$

$$\mathrm{Cov}(a,b) = -\frac{S_x}{S S_{tt}} \tag{15.2.21}$$

$$r_{ab} = \frac{\mathrm{Cov}(a,b)}{\sigma_a \sigma_b} \tag{15.2.22}$$

```
#include <math.h>
#include "nrutil.h"

void fit(float x[], float y[], int ndata, float sig[], int mwt, float *a,
    float *b, float *siga, float *sigb, float *chi2, float *q)
```

给定一组数据点 x[1..ndata],y[1..ndata],它们具有标准差 sig[1..ndata],通过求 $\chi^2$ 的极小值将这些数据点拟合成直线 $y=a+bx$。返回值 a、b 以及它们各自的不确定度 siga 和 sigb、chi 平方值 chi2、以及拟合优度概率值 q(拟合将使 $\chi^2$ 具有这么大或者更大)，如果在输入中 mwt=0，那么就认为数据点标准差不知道，q 将以1.0返回，chi2归一化为所有各点的单位标准差。

```
{
    float gammq(float a, float x);
    int i;
    float wt,t,sxoss,sx=0.0,sy=0.0,st2=0.0,ss,sigdat;
```

```
    *b=0.0;
    if (mwt) {                                          累加求和
        ss=0.0;
        for (i=1;i<=ndata;i++) {                         考虑权重
            wt=1.0/SQR(sig[i]);
            ss += wt;
            sx += x[i]*wt;
            sy += y[i]*wt;
        }
    } else {
        for (i=1;i<=ndata;i++) {                         没有权重
            sx += x[i];
            sy += y[i];
        }
        ss=ndata;
    }
    sxoss=sx/ss;
    if (mwt) {
        for (i=1;i<=ndata;i++) {
            t=(x[i]-sxoss)/sig[i];
            st2 += t*t;
            *b += t*y[i]/sig[i];
        }
    } else {
        for (i=1;i<=ndata;i++) {
            t=x[i]-sxoss;
            st2 += t*t;
            *b += t*y[i];
        }
    }
    *b /= st2;                                          求a, b σa 和σb
    *a=(sy-sx*(*b))/ss;
    *siga=sqrt((1.0+sx*sx/(ss*st2))/ss);
    *sigb=sqrt(1.0/st2);
    *chi2=0.0;                                          计算 χ2.
    if (mwt == 0) {
        for (i=1;i<=ndata;i++)
            *chi2 += SQR(y[i]-(*a)-(*b)*x[i]);
        *q=1.0;
        sigdat=sqrt((*chi2)/(ndata-2));                  对无权重数据，用chi2估计sig 并且
        *siga *= sigdat;                                 调整标准差
        *sigb *= sigdat;
    } else {
        for (i=1;i<=ndata;i++)
            *chi2 += SQR((y[i]-(*a)-(*b)*x[i])/sig[i]);
        *q=gammq(0.5*(ndata-2),0.5*(*chi2));              方程 (15.2.12).
    }
}
```

## 15.3 两个坐标数据都有误差的直线拟合

如果实验数据的测量误差不仅仅是 $y_i$ 存在误差，而 $x_i$ 也存在误差，则拟合直线模型

$$y(x) = a - bx \tag{15.3.1}$$

的任务将难得多，我们直接写出在这种情况下的 $\chi^2$ 优值函数

$$\chi^2(a,b) = \sum_{i=1}^{N} \frac{(y_i - a - bx_i)^2}{\sigma_{yi}^2 + b^2\sigma_{xi}^2} \tag{15.3.2}$$

其中 $\sigma_{xi}$ 和 $\sigma_{yi}$ 分别是第 $i$ 个点 $x$ 和 $y$ 的标准误差。等式(15.3.2)分母中的加权方差和，可以从两个方面来

理解，或者理解为在最小 $\chi^2$ 方向上每个数据点和斜率为 $b$ 的直线的方差，或者理解为随机变量 $x_i$ 和线性组合 $y_i - a - bx_i$ 的方差：

$$\mathrm{Var}(y_i - a - bx_i) = \mathrm{Var}(y_i) + b^2\mathrm{Var}(x_i) = \sigma_{yi}^2 + b^2\sigma_{xi}^2 \equiv 1/w_i \tag{15.3.3}$$

上式是 $N$ 个随机变量方差的平方和，并且每个随机变量的方差都已归一化，因此上面的式(15.3.3)是一个 $\chi^2$ 分布。

我们要寻找 $a$ 和 $b$ 的值以使式(15.3.2)最小。不幸的是，$b$ 在式(15.3.2)分母中的出现，使得求斜率等式 $\partial\chi^2/\partial b=0$ 是非线性的，但是对应于相应的截距条件 $\partial\chi^2/\partial a=0$，仍然是线性的，并且有

$$a = \left[\sum_i w_i(y_i - bx_i)\right] / \sum_i w_i \tag{15.3.4}$$

其中的 $w_i$ 由等式(15.3.3)定义。一个合理的方案是采用第十章中的方法(参看程序 **brent**)，通过选择 $b$ 的值求一般的一维函数的最小值，同时在每一步中保证对 $b$ 求最小值时，也对 $a$ 求最小值。

因为 $x_i$ 的误差是有限的，当 $b$ 为无穷大时以 $b$ 为变量的 $\chi^2$ 函数的最小值也是有限的，尽管这个值很大，因此角度 $\theta \equiv \arctan b$ 比 $b$ 更适于做斜率的参量。$\chi^2$ 的值从 $\theta$ 来看将是周期的，周期为 $\pi$(不是 $2\pi$)。如果数据点的 $\sigma_y$ 比较小，$\sigma_x$ 中等或很大，那么有可能在斜率 $\theta=0$ 处 $\chi^2$ 取最大值。在这种情况下，有可能存在两个 $\chi^2$ 最小值，一个在正斜率处，另一个在负斜率处。其中只有一个是全局的最小变量。因此对 $b$(或 $\theta$)的开始猜测值的选择是非常重要的。我们在下面采取的策略是对 $y_i$ 的值进行量化，以使它的方差等于 $x_i$ 的方差，然后根据从 $\sigma_{yi}^2+\sigma_{yi}^2$(量化的)导出的权重进行线性拟合(如同第15.2节)。如果数据点和线性模型确实非常相关的话，这将对 $b$ 产生一个非常好的猜测值。

要确定参数 $a$ 和 $b$ 的标准差 $\sigma_a$ 和 $\sigma_b$ 更加困难。我们在第15.6节中将看到，在适当的情况下，$a$ 和 $b$ 的标准差分别是"置信区域边界"在 $a$ 和 $b$ 轴上的投影，在"置信区域边界"上 $\chi^2$ 取比它的最小值大1的值，$\Delta\chi=1$。在第15.2节的线性情况下，这些投影遵从泰勒级数展开

$$\Delta\chi^2 \approx \frac{1}{2}\left[\frac{\partial^2\chi^2}{\partial a^2}(\Delta a)^2 + \frac{\partial^2\chi^2}{\partial b^2}(\Delta b)^2\right] + \frac{\partial^2\chi^2}{\partial a\,\partial b}\Delta a\Delta b \tag{15.3.5}$$

因为在目前情况下，$b$ 是非线性的，二阶导数的分析表达式非常繁，更重要的是，低阶项对 $\Delta\chi^2$ 经常给出一个比较差的近似。我们的策略是，通过调整斜率 $b$ 的值远离最小值来求解 $\Delta\chi^2=1$ 的根。在下面的程序中采用了一般的寻根程序 **zbrent**。有时可能一个根都没有。例如，如果所有的误差限非常大，以致于所有的数据点是彼此相容的。因此在精确确定一个根以前，划出一个求根区间就非常重要。

因为 $a$ 是在变化 $b$ 的每一步中得到最小值的，所以成功的数值求根必将导致 $\Delta a$ 值，它使得对应 $\Delta b$ 求 $\chi^2$ 最小值时 $\Delta\chi^2=1$。这(参看图15.3.1)直接给出了置信区域在 $b$ 轴上的切线投影，亦即 $\sigma_b$。但是它没有给出置信区域在 $a$ 轴上的切线投影。在图15.3.1中我们已经找到 $B$ 点。为了得到 $\sigma_a$，我们必须寻找点 $A$。几何上能保证：在置信区域近似是椭圆时，可以证明(参看图15.3.1)$\sigma_a^2=r^2+s^2$。$s$ 的值可以从图中的 $B$ 点而得到，而 $r$ 的值可以将等式(15.3.2)和(15.3.3)应用到 $\chi^2$ 的最小值(即图中的 0 点)而得到，给出

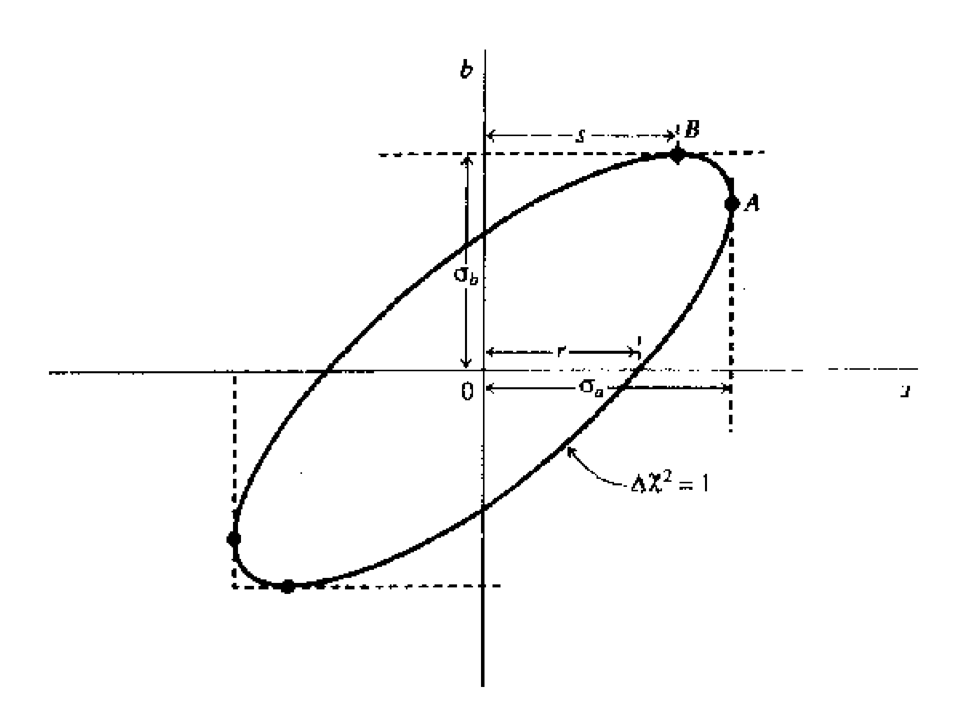
$$r^2 = 1/\sum_i w_i \tag{15.3.6}$$

实际上，因为 $b$ 能趋于无穷大，因此整个过程在$(a,\theta)$空间比在$(a,b)$空间更有意义。这实际就是下面程序的工作过程。但是习惯上总是返回 $a$ 和 $b$ 的标准差，而不是 $a$ 和 $\theta$，我们最后采用关系

$$\sigma_b = \sigma_\theta/\cos^2\theta \tag{15.3.7}$$

必须注意，如果 $b$ 和它的标准差都很大，以致于置信区域实际上将包括无穷大斜率，那么标准差 $b$ 没有很大意义。函数 **chixy** 通常仅被程序 **fitexy** 调用。但是，如果用户想要的话，可以自己探寻置信区域，这只要在初始调用一次 **fitexy** 后，重复调用 **chixy**(它的自变量是一个角度 $\theta$，而不是斜率 $b$)。

最后要注意的，和第15.6节相重复，也就是如果拟合优度不能被接受(被返回的概率太小)，标准差 $\sigma_a$ 和 $\sigma_b$ 当然是不可信的。在坏的情况下，你可能还要试着用一个常数因子标度 $x$ 和 $y$ 的误差限直到被返回的概率是可接受的(例如0.5)，以得到更合理的 $\sigma_a$ 和 $\sigma_b$ 值。




$B$ 点可以由变化斜率 $b$ 而同时求截距 $a$ 最小的方法得到。这将给出标准差 $\sigma_b$ 及 $s$ 的值。标准差 $\sigma_a$ 可以通过几何关系 $\sigma_a^2=s^2+r^2$ 而得到。

图15.3.1 参数 $a$ 和 $b$ 的标准差

```
#include <math.h>
#include "nrutil.h"
#define POTN 1.571000
#define BIG 1.0e30
#define PI 3.14159265
#define ACC 1.0e-3

int nn;                              与程序 chixy 传递的全局变量
float *xx, *yy, *sx, *sy, *ww, aa, offs;

void fitexy(float x[], float y[], int ndat, float sigx[], float sigy[],
    float *a, float *b, float *siga, float *sigb, float *chi2, float *q)
```

对 $x$ 和 $y$ 都具有误差的输入数据 x[1..ndat]和 y[1..ndat]进行数据拟合，它们各自对应误差在输入量 sigx[1..ndat]和 sigy[1..ndat]中。输出量是使拟合直线 $y=a+bx$ 的 $\chi^2$ 值最小的 a 和 b，这个最小 $\chi^2$ 以 chi2返回。$\chi^2$ 的概率以 q 返回，较小的 q 值表明较差的拟合(有时表明低估了的误差)。a 和 b 标准差以 siga 和 sigb 返回。这些结果在如下两种情况下将无意义(i)拟合较差，或者(ii)b 太大以致数据和垂直线(无穷大 b)一致。如果 siga 和 sigb 的返回值为 BIG，则数据将和 b 的所有值一致。

```
{
    void avevar(float data[], unsigned long n, float *ave, float *var);
    float brent(float ax, float bx, float cx,
        float (*f)(float), float tol, float *xmin);
    float chixy(float bang);
    void fit(float x[], float y[], int ndata, float sig[], int mwt,
        float *a, float *b, float *siga, float *sigb, float *chi2, float *q);
    float gammq(float a, float x);
    void mnbrak(float *ax, float *bx, float *cx, float *fa, float *fb,
        float *fc, float (*func)(float));
    float zbrent(float (*func)(float), float x1, float x2, float tol);
    int j;
    float swap,amx,amn,varx,vary,ang[7],ch[7],scale,bmn,bmx,d1,d2,r2,
```

```
        dum1,dum2,dum3,dum4,dum5;

    xx=vector(1,ndat);
    yy=vector(1,ndat);
    sx=vector(1,ndat);
    sy=vector(1,ndat);
    ww=vector(1,ndat);
    avevar(x,ndat,&dum1,&varx);                     // 寻找x和y变量、量化数据成为全局
    avevar(y,ndat,&dum1,&vary);                     //     变量以便和函数chixy进行传递
    scale=sqrt(varx/vary);
    nn=ndat;
    for (j=1;j<=ndat;j++) {
        xx[j]=x[j];
        yy[j]=y[j]*scale;
        sx[j]=sigx[j];
        sy[j]=sigy[j]*scale;
        ww[j]=sqrt(SQR(sx[j])+SQR(sy[j]));          // 在第一次试验拟合中考虑全部x和y
    }                                               //     的权重
    fit(xx,yy,nn,ww,1,&dum1,b,&dum2,&dum3,&dum4,&dum5);     // 对b试验拟合
    offs=ang[1]=0.0;                                // 根据参考数据点构造几个角度,并使b
    ang[2]=atan(*b);                                //     为一角度
    ang[4]=0.0;
    ang[5]=ang[2];
    ang[6]=PIO2;
    for (j=4;j<=6;j++) ch[j]=chixy(ang[j]);
    mnbrak(&ang[1],&ang[2],&ang[3],&ch[1],&ch[2],&ch[3],chixy);
    Bracket the χ² minimum and then locate it with brent.
    *chi2=brent(ang[1],ang[2],ang[3],chixy,ACC,b);
    *chi2=chixy(*b);
    *a=aa;
    *q=gammq(0.5*(nn-2),*chi2*0.5);                 // 计算χ²概率
    for (r2=0.0,j=1;j<=nn;j++) r2 += ww[j];         // 在最小值处保存权重和的倒数
    r2=1.0/r2;
    bmx=BIG;                                        // 现在在Δχ² = 1处寻找b的标准差
    bmn=BIG;
    offs=(*chi2)+1.0;
    for (j=1;j<=6;j++) {                            // 从保存的数据循环到括出的想要的根.
        if (ch[j] > offs) {                         //     注意倾斜角度的周期性
            d1=fabs(ang[j]-(*b));
            while (d1 >= PI) d1 -= PI;
            d2=PI-d1;
            if (ang[j] < *b) {
                swap=d1;
                d1=d2;
                d2=swap;
            }
            if (d1 < bmx) bmx=d1;
            if (d2 < bmn) bmn=d2;
        }
    }
    if (bmx < BIG) {                                // 调用zbrent寻根
        bmx=zbrent(chixy,*b,*b+bmx,ACC)-(*b);
        amx=aa-(*a);
        bmn=zbrent(chixy,*b,*b-bmn,ACC)-(*b);
        amn=aa-(*a);
        *sigb=sqrt(0.5*(bmx*bmx+bmn*bmn))/(scale*SQR(cos(*b)));
        *siga=sqrt(0.5*(amx*amx+amn*amn)+r2)/scale;     // a的误差多一项r2.
    } else (*sigb)=(*siga)=BIG;
    *a /= scale;                                    // 将标度的结果复原
    *b=tan(*b)/scale;
    free_vector(ww,1,ndat);
    free_vector(sy,1,ndat);
    free_vector(sx,1,ndat);
    free_vector(yy,1,ndat);
    free_vector(xx,1,ndat);
}
```

```c
#include <math.h>
#include "nrutil.h"
#define BIG 1.0e30

extern int nn;
extern float *xx, *yy, *sx, *sy, *ww, aa, offs;

float chixy(float bang)
程序 fitexy 的辅助函数,对于斜率 b=tan(bang),它返回(χ²−offs)值,被标度的数据和 offs 通过全局变量进行传递。
{
    int j;
    float ans,avex=0.0,avey=0.0,sumw=0.0,b;

    b=tan(bang);
    for (j=1;j<=nn;j++) {
        ww[j] = SQR(b*sx[j])+SQR(sy[j]);
        sumw += (ww[j] = (ww[j] == 0.0 ? BIG : 1.0/ww[j]));
        avex += ww[j]*xx[j];
        avey += ww[j]*yy[j];
    }
    if (sumw == 0.0) sumw = BIG;
    avex /= sumw;
    avey /= sumw;
    aa=avey-b*avex;
    for (ans = -offs,j=1;j<=nn;j++)
        ans += ww[j]*SQR(yy[j]-aa-b*xx[j]);
    return ans;
}
```

注意,看起来比较直接的、有关本节内容的文献,一般令人疑惑,有时还完全是错的。德敏(Deming)早期的处理是合理的,但是这处理依赖于泰勒展开,给出了不精确的误差估计。参考文献[2~4]是可靠的、更近代的,并以批判的态度对早期的工作进行了更一般的处理。约克(York)和里德(Read)对本节所处理的简单的直线情况进行了有用的讨论,但是后者的文献有些错误,这些错误在文献[7]中得到纠正。所有这些争论都被吸收到 Bayesians 的文献[8~10]中,他还有不同的观点。

## 参考文献和进一步读物:


Deming, W. E. 1943, *Statistical Adjustment of Data* (New York: Wiley), reprinted 1964 (New York: Dover). [1]

Jefferys, W. H. 1980, *Astronomical Journal*, vol. 85, pp. 177~181; see also vol. 95, p. 1299 (1988). [2]

Jefferys, W. H. 1981, *Astronomical Journal*, vol. 86, pp. 149~155; see also vol. 95, p. 1300 (1988). [3]

Lybanon, M. 1984, *American Journal of Physics*, vol. 52, pp. 22~26. [4]

York, D. 1966, *Canadian Journal of Physics*, vol. 44, pp. 1079~1086. [5]

Reed, B. C. 1989, *American Journal of Physics*, vol. 57, pp. 642~646; see also vol. 58, p. 189, and vol. 58, p. 1209. [6]

Reed, B. C. 1992, *American Joural of Physics*, vol. 60, pp. 59~62. [7]

Zellner, A. 1971, *An Introduction to Bayesian Inferences in Econometrics* (New York: Wiley), reprinted 1987 (Malabar, FL: R. E. Krieger Pub. Co). [8]

Gull, S.F. 1989, in *Maximum Entropy and Bayesian Methods*, J. Skilling, ed. (Boston: Kluwer). [3]
Jaynes, E. T. 1991, in *Maximum-Entropy and Bayesian Methods*, *Proc. 10th Int. Workshop*, W. T. Grandy, Jr., and L. H. Schick, eds. (Boston: Kluwer). [10]

## 15.4 一般的线性最小二乘方

第15.2节的一般推广是拟合一组数据$(x_i, y_i)$集的模型不只是 1 和 $x$ 的线性组合(也就是 $a+bx$),而是 $x$ 的任意 $M$ 个特定函数的线性组合。例如,$x$ 的函数可以是$1, x, \ldots, x^{M-1}$,这种情况下,它们的一般线性组合

$$y(x) = a_1 + a_2 x + a_3 x^2 + \ldots + a_M x^{M-1} \tag{15.4.1}$$

是一个 $M-1$阶的多项式。或者 $x$ 的函数是正弦和余弦,这种情况下它们的线性组合就是一个调和级数。

这种模型的一般形式是

$$y(x) = \sum_{k=1}^{M} a_k X_k(x) \tag{15.4.2}$$

其中 $X_1(x), \ldots, X_M(x)$是 $x$ 的任意函数,称为**基函数**。

注意,函数 $X_k(x)$可以是x 的非线性函数。我们这里讨论的线性是指模型和它的参数 $a_k$ 之间的关系。

对于这种线性模型我们推广前一节的讨论,定义优值函数

$$\chi^2 = \sum_{i=1}^{N} \left[ \frac{y_i - \sum_{k=1}^{M} a_k X_k(x_i)}{\sigma_i} \right]^2 \tag{15.4.3}$$

和前在一样,$\sigma_i$ 是第 $i$ 个数据点的测量误差(标准差),并且假设是已知的。如果测量误差不知道,那么和前面一样,让所有测量误差为常数值 $\sigma=1$。

我们将再次用求 $\chi^2$最小值的方法得出最佳参数。有几种不同的技巧可以求这个最小值。其中两种特别有用,在这一节我们都要讨论。为了引入和阐明它们之间的关系,我们需要引入一些符号。

设 $\mathbf{A}$ 为 $N\times M$ 阶的矩阵,它的元素由 $M$ 个基函数在 $N$ 个横坐标 $x_i$ 上的取值,以及 $N$ 个测量误差 $\sigma_i$ 计算求得,也就是如下的定义:

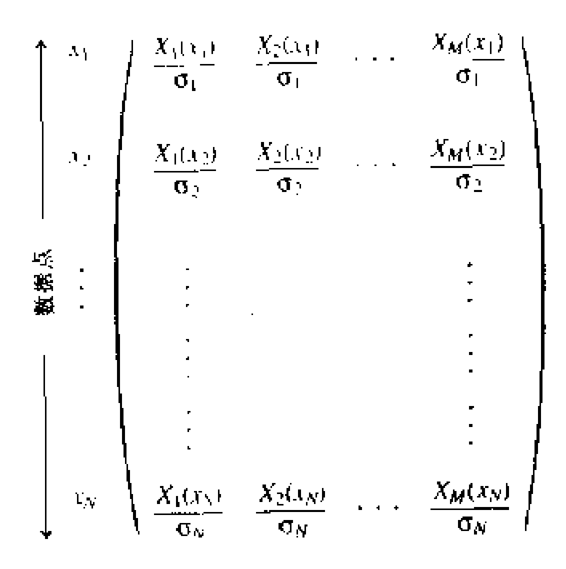
$$A_{ij} = \frac{X_j(x_i)}{\sigma_i} \tag{15.4.4}$$

矩阵 $\mathbf{A}$ 被称为拟合问题的**设计矩阵**。注意,通常情况下 $\mathbf{A}$ 的行数大于列数,即 $N\geqslant M$,因为数据点比要求模型参数多。(当然可以用两个数据点拟合一条直线,但不是非常有意义的事!)设计矩阵示于图15.4.1。

还通过下式定义长度为 $N$ 的向量 $\mathbf{b}$

$$b_i = \frac{y_i}{\sigma_i} \tag{15.4.5}$$

还定义一个长度为 M 的向量 $\mathbf{a}$,它的分量是要拟合的参数 $a_1, \ldots, a_M$。




由 $M$ 个基函数线性组合、对 $N$ 个数据点作最小二乘方拟合的设计矩阵。矩阵的元素包括由测量的自变量计算出来的基函数值，还考虑了已测量因变量的标准差，因变量的测量值没有被包括进这个矩阵。

图15.4.1

### 15.4.1 利用正规方程组求解

将式(15.4.3)中的 $\chi^2$ 分别对 $M$ 个参数 $a_k$ 求偏导并令之等于0，将求得 $\chi^2$ 的最小值。等式(15.1.7)在式(15.4.2)的模型情况下，将得到 $M$ 个方程

$$0 = \sum_{i=1}^{N} \frac{1}{\sigma_i^2}\left[y_i - \sum_{j=1}^{M} a_j X_j(x_i)\right] X_k(x_i) \qquad k = 1, \ldots, M \tag{15.4.6}$$

交换求和的顺序，可以将式(15.4.6)写成矩阵方程

$$\sum_{j=1}^{M} \alpha_{kj}\ a_j = \beta_k \tag{15.4.7}$$

其中

$$\alpha_{kj} = \sum_{i=1}^{N} \frac{X_j(x_i) X_k(x_i)}{\sigma_i^2} \qquad \text{或者等价} \qquad [\alpha] = \mathbf{A}^T \cdot \mathbf{A} \tag{15.4.8}$$

$[\alpha]$是一个 $M \times M$ 矩阵

$$\beta_k = \sum_{i=1}^{N} \frac{y_i X_k(x_i)}{\sigma_i^2} \qquad \text{或者等价} \qquad [\beta] = \mathbf{A}^T \cdot \mathbf{b} \tag{15.4.9}$$

$[\beta]$是一个长为 M 的向量。

方程组(15.4.6)或(15.4.7)称为最小二乘方问题的**正规方程组**。它们可以利用第二章中的标准方法求解得参数向量 $\mathbf{a}$，比较有名的是 $LU$ 分解法和回代法，回代法也称为高斯-约当(Gauss-Jordan)消元法。用矩阵的形式，正规方程可写为

$$[\alpha]\cdot\mathbf{a}=[\beta] \qquad 或者 \qquad (\mathbf{A}^T\cdot\mathbf{A})\cdot\mathbf{a}=\mathbf{A}^T\cdot\mathbf{b} \tag{15.4.10}$$

逆矩阵 $C_{jk}\equiv[\alpha]^{-1}_{jk}$，它和被估计参数 $\mathbf{a}$ 之可能的（更精确地说是标准的）不确定度有紧密联系。为了估计这些不确定度，考虑

$$a_j=\sum_{k=1}^{M}[\alpha]^{-1}_{jk}\beta_k=\sum_{k=1}^{M}C_{jk}\left[\sum_{i=1}^{N}\frac{y_iX_k(x_i)}{\sigma_i^2}\right] \tag{15.4.11}$$

根据式(15.2.7)估计参数 $a_j$ 的方差，可按下式计算

$$\sigma^2(a_j)=\sum_{i=1}^{N}\sigma_i^2\left(\frac{\partial a_j}{\partial y_i}\right)^2 \tag{15.4.12}$$

注意 $\alpha_{jk}$ 和 $y_i$ 是相互独立的，因此

$$\frac{\partial a_j}{\partial y_i}=\sum_{k=1}^{M}C_{jk}X_k(x_i)/\sigma_i^2 \tag{15.4.13}$$

结果我们得到

$$\sigma^2(a_j)=\sum_{k=1}^{M}\sum_{l=1}^{M}C_{jk}C_{jl}\left[\sum_{i=1}^{N}\frac{X_k(x_i)X_l(x_i)}{\sigma_i^2}\right] \tag{15.4.14}$$

上式括号内的项恰好是矩阵[$\alpha$]。因为它是矩阵[$C$]的逆矩阵，式(14.3.14)马上简单化为

$$\sigma^2(a_j)=C_{jj} \tag{15.4.15}$$

换言之，矩阵[$C$]的对角元是拟合参数 $a$ 的方差（不确定度的平方）。毫无疑问，[$C$]的非对角元 $C_{jk}$ 是 $a_j$ 和 $a_k$ 的协方差（参看15.2.10）；但我们将把这些问题的讨论推迟到第15.6节。

我们将利用解正规方程的方法，给出一个实现上面所讨论的一般线性最小二乘方的程序。因为我们要计算的不仅是待求的向量 $\mathbf{a}$，还包括协方差矩阵[$C$]，更方便的是采用高斯-约当消元法（第2.1节中 **gaussj** 程序）实现线性代数运算。在这种情况下的运算量不比 $LU$ 分解法大。但是如果不需要计算协方差矩阵，就可以转到 $LU$ 分解法，在线性代数中将减少1/3的计算量，此时不需计算逆矩阵。从理论上看，因为 $\mathbf{A}^T\cdot\mathbf{A}$ 是正定的，乔莱斯基分解是最有效的解正规方程的方法。但是在实际中大部分计算时间被用在循环数据以形成方程上了，所以高斯-约当方法是足够的了。

我们需要提醒注意的是，直接从正规方程得到的最小二乘问题的解对舍入误差十分敏感。一个更常用的可替代它的方法是设计矩阵 $\mathbf{A}$ 的 $QR$ 分解（第2.10节、第11.3节和第11.6节）。此方法对于我们在第15.2节最后所讨论的直线拟合非常重要，但是没有用到导出 QR 所需公式的运算。在这一节的后面，我们将讨论最小二乘方问题的其它难题，解决这些事难题的方法是奇异值分解法（SVD），我们将给出它的运算。已经证明 SVD 还能解决舍入误差问题，因此除了"简单"的最小二乘方问题外，我们建议采用这种方法。对于那些简单的问题，我们倾向于下面的程序，用它来解正规方程。

下面的程序的程序引入了一个在实际工作非常有用的窍门。通常我们要决定哪些模型参数须从数据组中得出，哪些参数设定为固定值，这些固定值可以是被某理论预计的，也可以是前一次实验测得的。[illegible]通常是具有"艺术"技巧的。因此必须有"冻结"和"不冻结"参数 $a_k$ 的方便手段。在下面的程序中，参数 $a_k$ 的总数目以 ma 表示（前面是用 $M$ 表示）。做为程序的输[illegible] [1..ma]，它们的分量要么是零，要么不是零（比如1）。是零则表示要求参数向量 a[1..ma]中相应元素保持固定在它们的输入值上。非零则表示要对参数进

行拟合。输出时,任何冻结的参数都有它们的方差,而它们的协方差在协方差矩阵中都置为0。

```
#include "nrutil.h"

void lfit(float x[], float y[], float sig[], int ndat, float a[], float ia[],
    int ma, float **covar, float *chisq, void (*funcs)(float, float [], int))
```

给定一组数据点x[1..ndat],y[1..ndat],各点的标准差由sig[1..ndat]给出,用求$\chi^2$最小值的方法确定某函数的a[1..ma]系数中全部或某些系数,这函数和系数a之间是线性关系,即 $y=\sum_i a_i \times afunc_i(x)$。输入数组ia[1..ma]中非零值项表示a中这些分量是需要拟合的,而零值表示a中这些分量应该保持固定在它们的输入值。程序的返回值为a[1..ma],$\chi^2$=chisq以及协方差矩阵covar[1..ma][1..ma]。(参数保持固定时将返回零协方差。)用户都必须提供一个函数程序**funcs**(x,afunc,ma),它返回在$x=x$处计算的ma个基函数值的数组afunc[1..ma]。

```
{
    void covart(float **covar, int ma, int ia[], int mfit);
    void gaussj(float **a, int n, float **b, int m);
    int i,j,k,l,m,mfit=0;
    float ym,wt,sum,sig2i,**beta,*afunc;

    beta=matrix(1,ma,1,1);
    afunc=vector(1,ma);
    for (j=1;j<=ma;j++)
        if (ia[j]) mfit++;
    if (mfit == 0) nrerror("lfit: no parameters to be fitted");
    for (j=1;j<=mfit;j++) {                  (对称)矩阵初始赋值
        for (k=1;k<=mfit;k++) covar[j][k]=0.0;
        beta[j][1]=0.0;
    }
    for (i=1;i<=ndat;i++) {                  对数据循环以累积计算正规方程的系数

        (*funcs)(x[i],afunc,ma);
        ym=y[i];
        if (mfit < ma) {                     排除对拟合函数已知部分的依赖关系
            for (j=1;j<=ma;j++)
                if (!ia[j]) ym -= a[j]*afunc[j];
        }
        sig2i=1.0/SQR(sig[i]);
        for (j=0,l=1;l<=ma;l++) {
            if (ia[l]) {
                wt=afunc[l]*sig2i;
                for (j++,k=0,m=1;m<=l;m++)
                    if (ia[m]) covar[j][++k] += wt*afunc[m];
                beta[j][1] += ym*wt;
            }
        }
    }
    for (j=2;j<=mfit;j++)                    根据对称性填充对角线以上部分
        for (k=1;k<j;k++)
            covar[k][j]=covar[j][k];
    gaussj(covar,mfit,beta,1);               矩阵求解
    for (j=0,l=1;l<=ma;l++)
        if (ia[l]) a[l]=beta[++j][1];        适合系数a的部分解
    *chisq=0.0;
    for (i=1;i<=ndat;i++) {                  求拟合的χ²值
        (*funcs)(x[i],afunc,ma);
        for (sum=0.0,j=1;j<=ma;j++) sum += a[j]*afunc[j];
        *chisq += SQR((y[i]-sum)/sig[i]);
    }
    covsrt(covar,ma,ia,mfit);                将协方差矩阵排序成拟合系数的真实顺序
    free_vector(afunc,1,ma);
    free_matrix(beta,1,ma,1,1);
}
```

上面程序中最后调用函数 **covsrt** 的目的是扩展协方差矩阵为 ma×ma 的协方差矩阵，根据正确的行号和列号排列，使“冻结”的变量的方差和协方差为 0。

函数 **covsrt** 如下。

```
#define SWAP(a,b) {swap=(a);(a)=(b);(b)=swap;}

void covsrt(float **covar, int ma, int ia[], int mfit)
  在存储器中扩展协方差矩阵 covar,以便考虑那些被保持固定的参数。(对后者,返回协方差为零)。
{
    int i,j,k;
    float swap;

    for (i=mfit+1;i<=ma;i++)
        for (j=1;j<=i;j++) covar[i][j]=covar[j][i]=0.0;
    k=mfit;
    for (j=ma;j>=1;j--){
        if (ia[j]){
            for (i=1;i<=ma;i++) SWAP(covar[i][k],covar[i][j])
            for (i=1;i<=ma;i++) SWAP(covar[k][i],covar[j][i])
            k--;
        }
    }
}
```

### 15.4.2 运用奇异值分解法求解

在某些情况中，正规方程对于解决最小二乘问题是完全足够的。但是，在很多情况下，正规方程非常接近奇异性。在解线性方程时经常会碰 零主元素（例如在 **gaussj** 中），在这种情况下，将得不到任何解。或者可能碰到一个非常小的主元素值，在这种情况下，拟合的参数 $a_k$ 将非常大，太灵敏而且不稳定，以致在对拟合函数进行估计时就不可能得到精确解。

这种情况为什么会发生呢？原因在于作为一个实验者，你不得不承认，数据不能清楚地将所提供的两个或多个基函数区分开。如果两个这样的函数，或者函数的两个不同的组合，恰巧它们对数据能进行一样好的或一样坏的拟合——则矩阵[α]不能分辨它们，使两列或两行完全重叠，形成奇异值。具有数学讽刺意味的是在最小二乘方问题中，超定的（数据点大于参数个数）和欠定的（存在参数的二义性组合）同时存在；但这还只是通常情况而已。在复杂问题中，这种二义性很难预先注意到。

现在考虑奇异值分解法，这将需重温第2.6节中的内容，这里不再重述。在超定系统的情况下，SVD 将得出最小二乘方意义下的最佳近似解，参看等式(2.6.10)。那正是我们所想要的。在欠定系统的情况下，SVD 得出一个解，它的值（对于我们来说就是 $a_k$ 的值）在最小二乘方意义下是最小的，参看等式(2.6.8)。这同样也是我们所要的：当某些基函数的组合和拟合不相干时，SVD 使这些组合得出一些小的无害的值，而不是上升为脆弱的无结果的无穷大。

用设计矩阵 **A**（等式(15.4.4)和向量 **b**（等式(15.4.5)）的形式，在(15.4.3)式中求 $\chi^2$ 的最小值可写为

$$\text{求}\ \mathbf{a}\ \text{使得}\ \chi^2 = |\mathbf{A}\cdot\mathbf{a} - \mathbf{b}|^2\ \text{取最小值} \tag{15.4.16}$$

和等式(2.6.9)相比，我们可以看出这恰好是程序 **svdcmp** 和 **svbksb** 所要求解的问题。由方程(2.6.12)所给出的解可重写成如下形式。如果根据等式(2.6.1)，将 **U** 和 **V** 引入 **A** 为

SVD 分解中，如同 **svdcmp** 中的所计算那样，则设向量 $\mathbf{U}_{(i)}$ $i=1,\ldots,M$ 表示 U 的一列(每一列都是长度为 $N$ 的向量)；还设向量 $\mathbf{V}_{(i)}$，$i=1,\ldots,M$ 表示 **V** 的一列(每列是长度为 $M$ 的向量)，则最小二乘方问题(15.4.16)的解(2.6.12)可写为

$$\mathbf{a} = \sum_{i=1}^{M}\left(\frac{\mathbf{U}_{(i)}\cdot\mathbf{b}}{w_i}\right)\mathbf{V}_{(i)} \tag{15.4.17}$$

其中 $w_i$ 如第2.6节中所示，是由 **svdcmp** 所计算的奇异值。

方程(15.4.17)表示拟合参数 **a** 是 V 之列向量的线性组合。其系数为 U 之列向量和数据之权重向量(15.4.5)的点积。已经证明拟合参数的标准误差(不严格地说，可能的误差)也是 **V** 列向量的线性组合，尽管这个证明超出了我们现在所讨论的范围。事实上，等式(15.4.17)可写成呈现出这些误差的形式：

$$\mathbf{a} = \left[\sum_{i=1}^{M}\left(\frac{\mathbf{U}_{(i)}\cdot\mathbf{b}}{w_i}\right)\mathbf{V}_{(i)}\right] + \frac{1}{w_1}\mathbf{V}_{(1)} \pm \cdots \pm \frac{1}{w_M}\mathbf{V}_{(M)} \tag{15.4.18}$$

在这里，每个±号后面是一个标准偏差。令人惊讶的是，尽管分解成这种形式，这些标准偏差是相互独立的(不相关的)。因此它们可以利用平方根形式加在一起。以后我们将会看到向量 $\mathbf{V}_{(i)}$ 是拟合参数 **a** 的误差椭球的主轴(看第15.6节)。

拟合参数 $a_j$ 的方差由下式给出

$$\sigma^2(a_j) = \sum_{i=1}^{M}\frac{1}{w_i^2}[\mathbf{V}_{(i)}]_j^2 = \sum_{i=1}^{M}\left(\frac{V_{ji}}{w_i}\right)^2 \tag{15.4.19}$$

这个结果应该和式(15.4.14)等价。和前面一样，对于协方差的公式应该毫不惊讶，这里我们可以不加证明地给出

$$\mathrm{Cov}(a_j,a_k) = \sum_{i=1}^{M}\left(\frac{V_{ji}V_{ki}}{w_i^2}\right) \tag{15.4.20}$$

在讨论这个小节时，我们曾提到由于遇到零主元，正规方程无效。但是到现在我们还没有说明 SVD 是如何克服这一困难的。答案在于：如果某一奇异值 $w_i$ 是零，那么在方程(15.4.18)中它的倒数将被置为 0，而不是无穷大(和前面等式(2.6.7)所讨论的相比)。这相当于给拟合参数 $a$ 加一个零倍基函数的线性组合，而不是某个随机大倍数的基函数的线性组合，这些基函数在拟合中是退化的。这简直太奇妙了。

还有，如果一个奇异值 $w_i$ 不是 0 但是非常小，同样必须定义它的倒数为 0，因为它的表现值可能是人为的舍入误差，而不是一个有意义的数字。一个似乎合理的问题是"怎样小的程度才算是小值?"在本节，我们认定所有的奇异值满足它和最大奇异值之比值小于机器精度 $\epsilon$ 的 $N$ 倍。(你可能赞同 $\sqrt{N}$ 或者某个常数，而不是 $N$ 做为倍数更好；这就开始涉及与硬件有关的问题了。)

还有另外一个原因使我们处理甚至是附加的奇异值，这些奇异值太大，以致于舍入误差毫无影响。奇异值分解法能使用户鉴别出，对减小数据集的 $\chi^2$ 值作用不大的那些变量的线性组合。对这些奇异值的处理就能极大地减小系数的可能误差，而对最小值 $\chi^2$ 的增加却可以忽略。在第15.6节我们进一步讨论如何鉴别和处理这些情况。在下面的程序中，可能发生这种处理的奇异值点将被说明。

一般说来，我们建议读者尽量使用 SVD 方法，而不用正规方程。SVD 的唯一不足在于它要求额外大小为 $N\times M$ 的数组来存储整个设计矩阵，这些存储空间被矩阵 U 重新写过。

SVD 还要求存储 $M\times M$ 的矩阵 $\mathbf{V}$，但这是用来代替在正规方程中同样大小的系数矩阵。SVD 比解正规方程明显地要慢；但是它的最大优点，即它永远不会无效(理论上)，这就弥补了它在运行速度上的不足。

在下面的程序中，矩阵 u、v 和向量 w 作为工作空间的输入。问题的逻辑尺度由 ndata 个数据点和 ma 个基函数(也即 ma 个拟合参数)决定。如果只关心拟合参数 a 的值，则输出的 u、v、w 所包含的信息毫无用处。如果要得到拟合参数的可能误差，则保存它们的值。

```
#include "nrutil.h"
#define TOL 1.0e-5

void   svdfit(float x[], float y[], float sig[], int ndata, float a[], int ma,
    float **u, float **v, float w[], float *chisq, void(**funcs)(float, float[], int))
    给定一组数据点x[1..ndata]及y[1..ndata]，它们的标准差由sig[1..ndata]给出，用求χ²最小值的方法来确定拟合
    函数 y = Σ a_i × afunc_i(x) 的系数a[1..ma]。在这里我们利用矩阵ma、采用奇异值分解法解ndata个数据(拟
    合)方程，正如第2.9节所述。数组u[1..ndata][1..ma]、v[1..ma][1..ma]和w[1..ma]为输入提供工作空间；在输
    出时，它们定义了奇异值分解，并且可用来得到协方差矩阵。程序返回值是ma个拟合参数a以及χ²值chisq。程序使
    用者必须提供一个程序funcs(x,afunc,ma)，它的返回值是在x=x处由ma个基函数计算所得的值，存放于数组a
    func[1..ma]中。
{
    void svbksb(float **u, float w[], float **v, int m, int n, float b[],
        float x[]);
    void svdcmp(float **a, int m, int n, float w[], float **v);
    int j,i;
    float wmax,tmp,thresh,sum,*b,*afunc;

    b=vector(1,ndata);
    afunc=vector(1,ma);
    for (i=1;i<=ndata;i++) {                累积计算拟合矩阵的系数。
        (*funcs)(x[i],afunc,ma);
        tmp=1.0/sig[i];
        for (j=1;j<=ma;j++) u[i][j]=afunc[j]*tmp;
        b[i]=y[i]*tmp;
    }
    svdcmp(u,ndata,ma,w,v);                 奇异值分解
    wmax=0.0;                               处理奇异值，在标题说明中给定TOL，由此
    for (j=1;j<=ma;j++)                         处起
        if (w[j] > wmax) wmax=w[j];
    thresh=TOL*wmax;
    for (j=1;j<=ma;j++)
        if (w[j] < thresh) w[j]=0.0;        …至此
    svbksb(u,w,v,ndata,ma,b,a);
    *chisq=0.0;                             估算chi-平方值
    for (i=1;i<=ndata;i++) {
        (*funcs)(x[i],afunc,ma);
        for (sum=0.0,j=1;j<=ma;j++) sum += a[j]*afunc[j];
        *chisq += (tmp=(y[i]-sum)/sig[i],tmp*tmp);
    }
    free_vector(afunc,1,ma);
    free_vector(b,1,ndata);
}
```

由上面程序所得的矩阵 v 和向量 w 将输入下面的短程序，就马上得到拟合参数 a 的方差和协方差，方差的平方根即标准差。此程序是对等式(15.4.20)的直接运算，同时遵从约定，等于 0 的奇异值点在拟合中已经分辨出来并且作了处理。

```
#include "nrutil.h"
```

```
void svdvar(float **v, int ma, float w[], float **cvm)
```
为了计算程序 **svdfit** 所得的 ma 个拟合参数的协方差矩阵 cvm[1..ma][1..ma]，可调用此程序，调用时将程序 **svdfit** 的输出结果，矩阵 v[1..ma][1..ma]及 w[1..ma]输入本程序
```
{
    int k,j,i;
    float sum, *wti;

    wti=vector(1,ma);
    for (i=1;i<=ma;i++) {
        wti[i]=0.0;
        if (w[i]) wti[i]=1.0/(w[i]*w[i]);
    }
    for (i=1;i<=ma;i++) {                              计算协方差矩阵(15.4.20)中和项
        for (j=1;j<=i;j++) {
            for (sum=0.0,k=1;k<=ma;k++) sum += v[i][k]*v[j][k]*wti[k];
            cvm[j][i]=cvm[i][j]=sum;
        }
    }
    free_vector(wti,1,ma);
}
```

### 15.4.3 例子

提醒注意的是，某些明显的非线性问题可以稍加变形而成为线性的。例如，一个带有两个参数 $a$ 和 $b$ 的指数函数模型

$$y(x) = a\exp(-bx) \tag{15.4.21}$$

可改写为

$$\log[y(x)] = c - bx \tag{15.4.22}$$

它关于参数 $c$ 和 $b$ 是线性的。(当然必须提醒注意，这种变换不能完整地将高斯误差转换成高斯误差)。

再请注意那些“非参数”情况，如同下式中

$$y(x) = a\exp(-bx-d) \tag{15.4.23}$$

参数 $a$ 和 $d$ 实际上是不可区分的。这是一个非常好的事例显示正规方程和 SVD 的区别，正规方程将碰到奇异值(无法解决)，而 SVD 将产生零奇异值。SVD 将根据“最小二乘方”原理在 $a$ 和 $d$ 之间建立一种平衡(或者，更倾向于通过取对数后得到它们等价的线性模型)。但是——这对于 SVD 无论在何时给出零奇异值都是正确的——我们建议最好还是通过分析鉴别出基函数组中的退化所在处，并在基函数组中做出适当的删除。

下面是要求用户所提供程序 **funcs** 的两个事例，第一个例子非常一般，用一组数据拟合普通的多项式。

```
void fpoly(float x, float p[], int np)
```
一个阶数为 np-1次多项式的拟合程序，它们的系数放在数组 p[1..np]中。
```
{
    int j;

    p[1]=1.0;
    for (j=2;j<=np;j++) p[j]=p[j-1]*x;
}
```

第二例子稍微有点特别，它是用一组数据点拟合 nl-1阶的勒让德多项式。

void fleg(float x,float pl[], int nl)
用如同第5.5节中的递推关系进行估值,得nl次勒让德多项式pl的拟合程序。

```
{
    int j;
    float twox, f2, f1, d;

    pl[1]=1.0;
    pl[2]=x;
    if (nl > 2) {
        twox=2.0*x;
        f2=x;
        d=1.0;
        for (j=3;j<=nl;j++) {
            f1=d++;
            f2 += twox;
            pl[j]=(f2*pl[j-1]-f1*pl[j-2])/d;
        }
    }
}
```

### 15.4.4 多维拟合

如果要测量的单变量 $y$ 是多个变量的函数——例如,变量为 $\mathbf{x}$ 向量,那么基函数将是一个向量 $X_1(\mathbf{x}),\ldots,X_M(\mathbf{x})$ 的函数,$\chi^2$ 优值函数现在是

$$\chi^2=\sum_{i=1}^{N}\left[\frac{y_i-\sum_{k=1}^{M}a_kX_k(\mathbf{x}_i)}{\sigma_i}\right]^2 \tag{15.4.24}$$

前面的所有讨论都不变,只是 $x$ 被 $\mathbf{x}$ 代替。实际上,如果能容忍程序的切割,则可以直接用上面程序,而不加任何改动:在 **lfitt** 和 **svdfit** 中唯一要用到的数组 x[i]中的每一元素都返回到用户提供的程序 **funs**,它必将给出在那点的基函数的值。如果在调用 **lfit** 或 **svdfit** 前使 x(i)=i,并独立地将数据点(也就是全局变量)构成的真实向量值提供给 funcs,则 funcs 就能在开始它的工作之前由虚构的 x[i]变为真实的数据点。

## 15.5 非线性模型

我们现在考虑,模型和 $M$ 个未知参数集 $a_k(k=1,2,\ldots,M)$ 之间是**非线性**关系的拟合。我们使用和前一节相同的方法,也就是定义 $\chi^2$ 优值函数,通过求它的最小值确定最佳拟合参数。但是在非线性关系时,求最小值必须迭代进行。对参数给定一个初始值,我们设计一个改善这个初始实验解的过程,这个过程接着不断重复进行,直到 $\chi^2$ 值不再增长(或者不再明显增长)为止。

这个问题和我们在第十章已经解决的求一般非线性函数极小化的问题有何不同呢?表面上有不同,但不是完全不同:当非常接近极小值时,我们希望 $\chi^2$ 函数趋近于二次型,我们可以将它写为

$$\chi^2(\mathbf{a})\approx\gamma-\mathbf{d}\cdot\mathbf{a}+\frac{1}{2}\mathbf{a}\cdot\mathbf{D}\cdot\mathbf{a} \tag{15.5.1}$$

其中,$\mathbf{d}$ 是长度为 $M$ 的向量,而 $\mathbf{D}$ 是 $M\times M$ 的矩阵(和方程(10.6.1)比较)。如果这种近似

是一个很好的近似，则我们知道怎样从当前的实验参数值 $\mathbf{a}_{cur}$ 一步跳到极小值的 $\mathbf{a}_{min}$，也就是

$$\mathbf{a}_{min} = \mathbf{a}_{cur} + \mathbf{D}^{-1}[-\nabla\chi^2(\mathbf{a}_{cur})] \tag{15.5.2}$$

（和式(10.7.4)相比）。

另一方面，式(15.5.1)对于在 $\mathbf{a}_{cur}$ 处进行极小化的函数形状是一个比较差的局域近似。在这种情况下，我们首先要做的是，沿负梯度的方向走一步，如同在最速下降法中那样（第10.6节）。换言之，即

$$\mathbf{a}_{next} = \mathbf{a}_{cur} - 常数 \times \nabla\chi^2(\mathbf{a}_{cur}) \tag{15.5.3}$$

其中常数要足够小，以保证不偏离下降方向。

为了利用式(15.5.2)或者(15.5.3)，我们必须能够计算在任意参数集 $\mathbf{a}$ 上 $\chi^2$ 函数的梯度。为了利用式(15.5.2)，我们也必须计算矩阵 $\mathbf{D}$，它是在 $\mathbf{a}$ 取任意值时，$\chi^2$ 优值函数的二阶导数矩阵（海赛矩阵）。

这里所讨论的和第十章的重要区别在于：在这里我们没有直接计算海赛矩阵的方法。我们只能估算需要极小化的函数以及（在某些情况下）它的梯度，因此我们必须求助于迭代方法，不仅仅是因为我们的函数是非线性，而且是为了我们要构造海赛矩阵的内容。第10.7节和第10.6节中已提到两种不同构造这些内容的方法。

在这里，情况要简单得多。我们精确地知道 $\chi^2$ 的形式，因为它建立在我们自己已经规定的模型函数的基础上，因此海赛(Hessian)矩阵对我们来说是已知的。只要我们愿意，我们便可无约束地采用等式(15.5.2)。使用式(15.5.3)的唯一原因在于，式(15.5.2)不能成功地改善拟合，并标志着式(15.5.1)做为一个局域近似的不可行。

### 15.5.1 梯度和海赛矩阵的计算

需要拟合的模型是

$$y = y(x;\mathbf{a}) \tag{15.5.4}$$

$\chi^2$ 优值函数是

$$\chi^2(\mathbf{a}) = \sum_{i=1}^{N}\left[\frac{y_i - y(x_i;\mathbf{a})}{\sigma_i}\right]^2 \tag{15.5.5}$$

$\chi^2$ 对参数 $\mathbf{a}$ 的梯度，在 $\chi^2$ 取极小值时为 0，它的各个分量是

$$\frac{\partial\chi^2}{\partial a_k} = -2\sum_{i=1}^{N}\frac{[y_i - y(x_i;\mathbf{a})]}{\sigma_i^2}\frac{\partial y(x_i;a)}{\partial a_k} \qquad k = 1,2,\ldots,M \tag{15.5.6}$$

再求一次导数得到

$$\frac{\partial^2\chi^2}{\partial a_k\partial a_l} = 2\sum_{i=1}^{N}\frac{1}{\sigma_i^2}\left[\frac{\partial y(x_i;\mathbf{a})}{\partial a_k}\frac{\partial y(x_i;\mathbf{a})}{\partial a_l} - [y_i - y(x_i;\mathbf{a})]\frac{\partial^2 y(x_i;\mathbf{a})}{\partial a_l\partial a_k}\right] \tag{15.5.7}$$

可以通过如下定义而去掉因子2

$$\beta_k \equiv -\frac{1}{2}\frac{\partial\chi^2}{\partial a_k} \qquad \alpha_{kl} \equiv \frac{1}{2}\frac{\partial^2\chi^2}{\partial a_k\partial a_l} \tag{15.5.8}$$

和式(15.5.2)中的 $\mathbf{D}$ 的关系是 $[\alpha] = \frac{1}{2}\mathbf{D}$，利用式(15.5.8)可以把式(15.5.2)重写成一组线性方程组

$$\sum_{l=1}^{M} \alpha_{kl} \delta a_l = \beta_k \tag{15.5.9}$$

这个方程组解出增量 $\delta a_l$，和当前的近似相加给出下一步的近似。在最小二乘方问题的文献中，矩阵$[\alpha]$通常被称为**曲率矩阵**，等于海赛矩阵的二分之一。

等式(15.5.3)，即最速下降等式，可写为

$$\delta a_l = \text{常数} \times \beta_l \tag{15.5.10}$$

注意，海赛矩阵(15.5.7)的分量$\alpha_{kl}$依赖于基函数对它们参数的一阶及二阶导数。某些处理过程常常不加评注地忽略了二阶导数。我们也忽略二阶导数，但是稍加评注后才这样做。

当式(15.5.6)中的梯度依赖于导数$\partial y_i/\partial a_k$时，二阶导数便发生了，因为对梯度的微分必定包括象$\partial^2 y_i/\partial a_l \partial a_k$这里的项。当二阶微分项是0时自然可以去掉(例如在方程(15.4.8)的线性情况下)，或者和一阶导数项相比足够小以致于可以忽略。在实际应用中，还有另外一个因为小而可忽略的可能情况：在方程(15.5.7)中，乘以二阶导数的项是$[y_i - y(x_i;\mathbf{a})]$。对于一个成功的模型来说，这一项应当是每一点的随机测量误差。这个误差可正可负，并且一般应该和模型无关。因此，当对 $i$ 累加求和时，二阶导数项就相互抵消了。

在拟合模型比较差，或者在模型数据中，不存在通过反号补偿而抵消的数据点时，若把二阶导数项包括进来，则会导致不稳定性。从现在起，我们将一直用下式作为$\alpha_{kl}$的定义

$$\alpha_{kl} = \sum_{i=1}^{N} \frac{1}{\sigma_i^2}\left[\frac{\partial y(x_i;\mathbf{a})}{\partial a_k} \frac{\partial y(x_i;\mathbf{a})}{\partial a_l}\right] \tag{15.5.11}$$

这个表达式和它在线性问题的同类项(15.4.8)非常相似。必须明白$[\alpha]$是无足轻重的，它对于参数 $\mathbf{a}$ 的最终结果毫无影响，只是影响获得此结果的迭代过程。$\chi^2$取最小值的条件，即对所有的 $k$ 存在 $\beta_k=0$，是和$[\alpha]$如何定义毫无关系的。

## 15.5.2 勒温伯格-马阔特方法

在勒温伯格(Levenberg)的一个早期建议的启示下，马阔特(Marquardt)[1]提出了一个非常巧妙的方法，它是在逆海赛矩阵方法(15.5.9)的极限和最速下降法(15.5.10)之间进行平滑调和。后一方法在远离最小值时运用，当接近最小值时它逐渐切换到前一方法。这种**勒温伯格-马阔特方法**(也称**马阔特方法**)在实际应用中非常奏效，并且已经成为非线性最小二乘方问题的标准。

这种方法基于两个基本的但是非常重要的洞察。考虑方程(15.5.10)中的"常数"，它应该是什么？大小数量级有多大？由什么来确定它的尺度大小？在梯度中没有关于这些答案的任何信息。梯度只说明了一个坡度，而没有说明这个坡度延伸到多远。马阔特首先洞察到，海赛矩阵的各个分量，尽管在一些精确场合是无用处的，但却对问题的尺度大小及数量级提供了一些信息。

量 $\chi^2$是无量纲的，也就是说它是一个纯粹的数；这可以从它的定义式(15.5.5)中明显看出。另一方面，$\beta_k$ 是由$1/a_k$ 量度的，它很可能有量纲，例如为 $\text{cm}^{-1}$，或者千瓦小时，或者其它形式的单位(实际上，$\beta_k$ 的各个分量可能有不同的量纲!)。因此在 $\beta_k$ 和 $\delta a_k$ 之间的比例常数必须具有量纲 $a_k^2$。检查一下$[\alpha]$的各个分量，会发现只有一个量明显地具有这种量纲，那就是$1/\alpha_{kk}$，即对角元的倒数。因此这些对角元确定了"常数"的尺度。但是这种尺度可能本身

就很大，因此我们引入一个人为的因子 $\lambda$，而将它除以常数（$\lambda$ 无量钢），并尽可能置 $\lambda>>1$ 以减少步长。换言之，用下式代替方程(15.5.10)：

$$\delta a_l = \frac{1}{\lambda \alpha_{ll}} \beta_l \quad \text{或} \quad \lambda \alpha_{ll} \delta a_l = \beta_l \tag{15.5.12}$$

上式要求 $\alpha_{ll}$是正的，这可以由定义式(15.5.11)保证——这也是采用式(15.5.11)的另一个原因。

马阔特的第二个洞察在于，等式(15.5.12)和(15.5.9)可以结合起来，如果我们按照如下规定定义一个新矩阵 $\alpha'$

$$\begin{aligned} \alpha'_{jj} &\equiv \alpha_{jj}(1+\lambda) \\ \alpha'_{jk} &\equiv \alpha_{jk} \qquad (j \neq k) \end{aligned} \tag{15.5.13}$$

那么式(15.5.12)和(15.5.9)均可用下式代替

$$\sum_{l=1}^{M} \alpha'_{kl} \delta a_l = \beta_k \tag{15.5.14}$$

当 $\lambda$ 非常大时，矩阵 $\alpha'$ 强制成以**对角元为主**的，此时等式(15.5.14)和(15.5.12)几乎等价。另一方面，当 $\lambda$ 趋近于零时，等式(15.5.14)便回到了式(15.5.9)。

对拟合参数 **a** 给定初始值后，我们建议采用如下的马阔特步骤

- 计算 $\chi^2(\mathbf{a})$。
- 取一个适中的 $\lambda$ 值，比如 $\lambda=0.001$。
- (†)解线性方程(15.5.14)得到 $\delta\mathbf{a}$，并且计算 $\chi^2(\mathbf{a}+\delta\mathbf{a})$。
- 如果 $\chi^2(\mathbf{a}+\delta\mathbf{a}) \geqslant \chi^2(\mathbf{a})$，则将 $\lambda$ 扩大 10 倍(或者其它的具有意义的倍数)，再回到(†)。
- 如果 $\chi^2(\mathbf{a}+\delta\mathbf{a}) < \chi^2(\mathbf{a})$，将 $\lambda$ 减小 10 倍，将 $\mathbf{a}+\delta\mathbf{a}$ 代替 **a**，再回到(†)。

我们还必须知道终止迭代的条件。收敛后继续进行迭代是不必要的，也是浪费的，因为求最小值充其量是对参数 **a** 的统计估计(收敛的程度决定于机器的精度以及舍入误差的极限)。正如我们将在第15.6节中所看到的那样，当 $\chi^2$的变化量$<<1$ 时，在统计的角度上参数的变化是毫无意义的。

进一步，我们经常发现极小值位于一个复杂的拓扑形状的平谷处，参数围绕此极小值徘徊。其原因在于马阔特方法是对正规方程(第15.4节)的推广，因此它也遇到和正规方程一样的问题，即在极小值处的退化。由于零主元也有可能导致的直接失败，但可能性不大。更经常遇到的情况是，一个小的主元将会产生一个大的补值而被排斥，$\lambda$ 的值随之增加。对于足够大的 $\lambda$，矩阵$[\alpha']$完全是正定的，不可能有很小的主元。因此这种方法的确能倾向于远离零主元，但付出的代价是，在一个下降坡度不大的谷地处做最速下降，从而使趋近于最小值比较慢。

从这些考虑我们建议，在实际中，在 $\chi^2$减少很小量的第一次或第二次时停止迭代，例如 $\chi^2$绝对值的减小量小于0.1，或者(以防舍入误差阻止获得这个精度)某些分数值如$10^{-3}$。不要在 $\chi^2$增加的那一步停止迭代：这只说明 $\lambda$ 还没有调整得非常合适。

一旦可接受的的极小值找到了，就可以令 $\lambda=0$，并计算矩阵

$$[C] \equiv [\alpha]^{-1} \tag{15.5.15}$$

如同前而所讨论的一样，上式可用来计算拟合参数 **a** 的协方差矩阵(参看下节)。

下面的一对函数程序准确地体现了非线性参数估计的马阔特方法。程序的大部分结构和在第15.4节中所用的 **lfit** 一致。特别要注意的是，在数组 ia[1..ma]中必须输入分量值为1 或零，它们分别对应于数组 a[1..ma]中相应的参数值是需要被拟合的，还是需要保持固定在它们的输入值。

程序 **mrqmin** 执行马阔特方法的一个迭代。在第一次调用它时令 alamda<0，它标志着程序的开始。alamda 首先设定后，在这个设定值下的调用将作为下次迭次的 $\lambda$ 值；a 和 chisq 作为迄今找到的最佳估计参数以及对应的 $\chi^2$ 返回。当你对收敛程度满意时，在进行最后一次调用前令 alamda 为 0。矩阵 alpha 和 covar（在所有先前的调用中它们被用作工作空间），就是收敛参数值的曲率矩阵和协方差。参量 alpha、a 和 chisq 在两次调用之间不能变化，alamde 除了在最后一次调用时设为零外，任何其它两次调用之间也保持不变。当遇到使 $\chi^2$ 增加的一个步骤时，chisq 和 a 返回它们的输入值（最佳值），但是 almada 将设置为一个增大的值返回。

程序 **mrqmin** 要调用程序 **mrqcof** 用来计算矩阵[$\alpha$]（参看等式15.5.11）和向量 $\beta$（等式15.5.6和等式15.5.8）。在 **mrqcof** 中又要调用用户提供的程序 **funcs**(x,a,y,dyda)，它用来计算输入值 **x** $\equiv x_i$ 及 $a \equiv$ **a** 时的模型函数值 y $\equiv y(x_i,\mathbf{a})$以及微分向量 duda $\equiv \partial y/\partial a_k$。

```
#include "nrutil.h"

void mrqmin(float x[], float y[],float sig[],int ndata, float a[], int ia[],
    int ma, float **covar, float **alpha, float *chisq,
    void (*funcs)(float, float [], float *, float [], int), float *alamda)
```

采用勒温伯格—马阔特方法来约化一组数据点 x[1..ndata]、y[1..ndata]和某一函数拟合的 $\chi^2$值，这些数据点具有标准差为 sig[1..ndata]，而非线性函数依赖的 ma 个系数为 a[1..ma]。输入数组 ia[1..ma]其非零项表明数组 a 中它些分量需要进行拟合，而零项表明 a 中那些分量应该保持固定在它们的输入量。程序返回当前参数 a[1..ma]的最佳拟合值和 $\chi^2$=chisq 值。在大多数迭代中，数组 covar[1..ma][1..ma]、alpha[1..ma][1..ma]用作为工作空间。用户提供一个子程序函数 funcs(x,a,yfit,dyda,ma)，它能计算拟合函数 yfit，以及它关于拟合参数 $a$ 在 $x$ 处的导数 dyda[1..ma]。第一次调用时，对参数 a 提供一个初始猜测值，并初始赋值 alamda<0（然后置 alamda=0.001）。如果一步成功，则 chisq 变得更小和 alamda 下降到1/10。如果一步失败，则 alamda 增加到10倍。必须反复调用本程序，直到达到收敛，然后，最终用 alamda=0调用一次本程度，使得在 covar[1..ma][1..ma]中返回协方差矩阵，alpha 返回曲率矩阵。（所保持固定的参数返回零协方差）。

```
{
    void covsrt(float **covar, int ma, int ia[], int mfit);
    void gaussj(float **a, int n, float **b, int m);
    void mrqcof(float x[], float y[], float sig[], int ndata, float a[],
        int ia[], int ma, float **alpha, float beta[], float *chisq,
        void (*funcs)(float, float [], float *, float [], int));
    int j,k,l,m;
    static int mfit;
    static float ochisq,*atry,*beta,*da,**oneda;

    if (*alamda < 0.0) {                    初始赋值
        atry=vector(1,ma);
        beta=vector(1,ma);
        da=vector(1,ma);
        for (mfit=0,j=1;j<=ma;j++)
            if (ia[j]) mfit++;
        oneda=matrix(1,mfit,1,1);
        *alamda=0.001;
        mrqcof(x,y,sig,ndata,a,ia,ma,alpha,beta,chisq,funcs);
        ochisq=(*chisq);
        for (j=1;j<=ma;j++) atry[j]=a[j];
```

```
    }
    for (j=0,l=1;l<=ma;l++) {            通过增加对角元，改变线性化的拟合矩阵
        if (ia[l]) {
            for (j++,k=0,m=1;m<=ma;m++) {
                if (ia[m]) {
                    k++;
                    covar[j][k]=alpha[j][k];
                }
            }
            covar[j][j]=alpha[j][j]*(1.0+(*alamda));
            oneda[j][1]=beta[j];
        }
    }
    gaussj(covar,mfit,oneda,1);          矩阵求解
    for (j=1;j<=mfit;j++) da[j]=oneda[j][1];
    if (*alamda == 0.0) {                已收敛，计算协方差矩阵
        covsrt(covar,ma,ia,mfit);
        free_matrix(oneda,1,mfit,1,1);
        free_vector(da,1,ma);
        free_vector(beta,1,ma);
        free_vector(atry,1,ma);
        return;
    }
    for (j=0,l=1;l<=ma;l++)              试验成功了吗
        if (ia[l]) atry[l]=a[l]+da[++j];
    mrqcof(x,y,sig,ndata,atry,ia,ma,covar,da,chisq,funcs);
    if (*chisq < ochisq) {               成功则接收新的解
        *alamda *= 0.1;
        ochisq=(*chisq);
        for (j=0,l=1;l<=ma;l++) {
            if (ia[l]) {
                for (j++,k=0,m=1;m<=ma;m++) {
                    if (ia[m]) {
                        k++;
                        alpha[j][k]=covar[j][k];
                    }
                }
                beta[j]=da[j];
                a[l]=atry[l];
            }
        }
    } else {                             失败则增加alamda和返回
        *alamda *= 10.0;
        *chisq=ochisq;
    }
}
```

注意其中使用了第15.4节中程序 **covsrt**。这仅仅是为了将协方差矩阵 covar 重排成所有 ma 个参数的顺序。上面程序中也使用了以下函数。

```
#include "nrutil.h"

void mrqcof(float x[], float y[],float sig[], int ndata, float a[], int ia[],
    int ma, float **alpha, float beta[], float *chisq,
    void (*funcs)(float, float [], float *, float [], int))
```
本程序被程序 **mrqmin** 用来计算式(15.5.8)中线性化了的拟合矩阵 alpha，和向量 beta，以及计算 $\chi^2$。
```
{
    int i,j,k,l,m,mfit=0;
    float ymod,wt,sig2i,dy,*dyda;

    dyda=vector(1,ma);
    for (j=1;j<=ma;j++)
        if (ia[j]) mfit++;
    for (j=1;j<=mfit;j++) {              (对称的)初始赋值
```

```
        for (k=1;k<=j;k++) alpha[j][k]=0.0;
        beta[j]=0.0;
    }
    *chisq=0.0;
    for (i=1;i<=ndata;i++) {            对所有数据循环求和
        (*funcs)(x[i],a,&ymod,dyda,ma);
        sig2i=1.0/(sig[i]*sig[i]);
        dy=y[i]-ymod;
        for (j=0,l=1;l<=ma;l++) {
            if (ia[l]) {
                wt=dyda[l]*sig2i;
                for (j++,k=0,m=1;m<=l;m++)
                    if (ia[m]) alpha[j][++k] += wt*dyda[m];
                beta[j] += dy*wt;
            }
        }
        *chisq += dy*dy*sig2i;              求找χ²
    }
    for (j=2;j<=mfit;j++)                填补对称的一边
        for (k=1;k<j;k++) alpha[k][j]=alpha[j][k];
    free_vector(dyda,1,ma);
}
```

## 15.5.3 例子

下面的函数 **fgauss** 是一个要用户提供的函数程序 **funcs** 的示例。应用上面的程序 **mrqmin**(同时要用到 **mrqcof**、**covsrt** 和 **gaussj**)它可用来拟合下面的模型

$$y(x) = \sum_{k=1}^{K} B_k \exp\left[-\left(\frac{x-E_k}{G_k}\right)^2\right] \tag{15.5.16}$$

它是 $K$ 个高斯函数的和，每个函数具有可变的位置、振幅和宽度。我们把各参数按 $B_1$,$E_1$,$G_1$,$B_2$,$E_2$,$G_2$,…,$B_k$,$E_k$,$G_k$ 的顺序存储。

```
#include <math.h>

void fgauss(float x, float a[], float *y, float dyda[], int na)
    y(x;a)是na/3个高斯函数式(15.5.16)的和。每个高斯函数的振幅、中心位置以及宽度连续存贮在a中：a[i]=B_k,a
    [i+1]=E_k,a[i+2]=G_k,k=1,...,na/3。数组的维数为a[1..na],dyda[1..na]。
{
    int i;
    float fac, ex, arg;

    *y=0.0;
    for (i=1;i<=na-1;i+=3) {
        arg=(x-a[i+1])/a[i+2];
        ex=exp(-arg*arg);
        fac=a[i]*ex*2.0*arg;
        *y += a[i]*ex;
        dyda[i]=ex;
        dyda[i+1]=fac/a[i+2];
        dyda[i+2]=fac*arg/a[i+2];
    }
}
```

## 15.5.4 非线性最小二乘方法的更先进的方法

勒温伯格-马阔特算法可以认为是，一种模型-置信区域方法应用于在最小二乘方函数之特殊情况中求极小值(参阅第9.7节和参考文献[2])。白摩瑞(More)[3]编制的这类代码可

以在 MINPACK[4]中找到。另外一种非线性最小二乘方算法，保留了我们在勒温伯格-马阔特方法中放弃的二阶导数项，任何时候保留这项将更好。这些方法被称为："全牛顿型"方法，并且认为它比勒温伯-马阔特方法更稳健，也更复杂。这些方法的一个实现是代码 NL2SOL[5]。

参考文献和进一步读物：


Marquardt, D.W. 1963, *Journal of the Society for Industrial and Applied Mathematics*, vol. 11, pp. 431～441. [1]

Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* (Englewood Cliffs, NJ: Prentice-Hall). [2]

More, J.J. 1977, in *Numerical Analysis*, Lecture Notes in Mathematics, vol. 630, G.A. Watson, ed. (Berlin: Springer-Verlag), pp. 105～116. [3]

More, J.J., Garbow, B.S., and Hillstrom, K.E. 1980, *User Guide for MINPACK-1*, Argonne National Laboratory Report ANL-80～74. [4]

Dennis, J.E., Gay, D.M, and Welsch, R.E. 1981, *ACM Transactions on Mathematical Software*, vol. 7, pp. 348-368; *op. cit.*, pp. 369～383. [5]


## 15.6 被估模型参数的置信界限

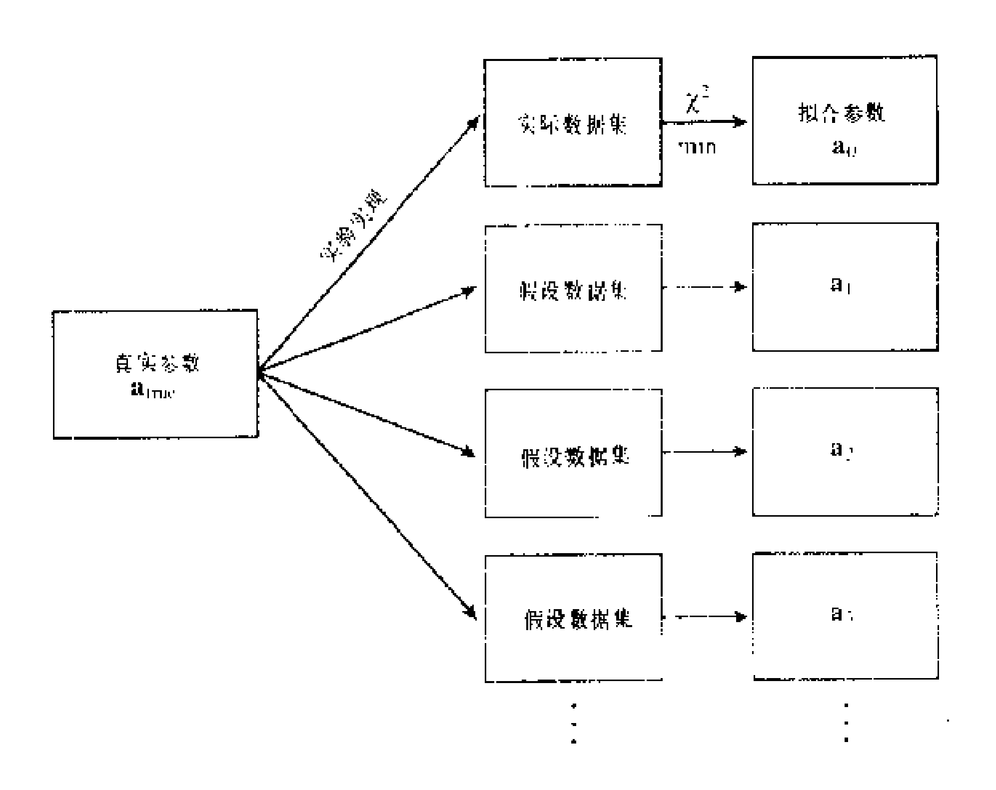
在这一章里，我们已经好几次论述 $M$ 个被估参数 $\mathbf{a}$ 的标准误差或不确定度。我们不仅给出了每对参数之间的协方差公式（等式(15.2.10)；在后面又用等式(15.4.15)、(15.4.20)及(15.5.15)）给出了单个参数的标准差或方差公式（等式(15.2.9)、(15.4.15)及(15.4.19)式）。

在这一节，我们要更明确地表述这些量的不确定度的精确含义，还要对如何定量地估计拟合参数的置信界限进行深入的讨论。这个问题的讨论涉及到一些技巧，甚至会感到一点困惑，因此我们将尽可能做精确描述，即使我们未做任何证明。

图15.6.1呈现了"测量"一组参数的实验的概念性结构。某些基本参数(集)的真实值 $\mathbf{a}_{true}$ 只有上帝才知道，实验也是不可能知道的。真实的参数是通过统计实现的，如同测量数据那样，带有随机测量误差，我们将用 $\mathcal{D}_{(0)}$ 标记数据集。数据集 $\mathcal{D}_{(0)}$ 对实验者来说是可知的。他或者她用数据拟合一个模型，通过求 $\chi^2$ 最小值或者其它的技巧，获得参数测量值即拟合值，在这里我们用 $\mathbf{a}_{(0)}$ 标记。

因为测量误差具有随机分量，$\mathcal{D}_{(0)}$ 不是真实参数值 $\mathbf{a}_{true}$ 的唯一实现。相反，存在真实参数的无穷多个其它实现，我们可以把它们作为"假设的数据集"，其中每一组都可能已是一组测量结果，但恰巧又不是。让我们把这些数据集用 $\mathcal{D}_{(1)}$、$\mathcal{D}_{(2)}$，……标记。每一个数据集如果实现了的话，将分别给出稍微有些不同的拟合参数集 $\mathbf{a}_{(1)}$，$\mathbf{a}_{(2)}$，……因此，这些参数集 $\mathbf{a}_{(i)}$ 将在所有参数集 $\mathbf{a}$ 构成的 $M$ 维空间中呈现出一种概率分布。实际测量到的一组参数 $\mathbf{a}_{(0)}$ 是从这个分布中导出的一个结果。

比 $\mathbf{a}_{(i)}$ 的概率分布更有兴趣的是 $\mathbf{a}_{(i)}-\mathbf{a}_{true}$ 差的分布。这个分布和前面的所讨论的分布所不同之处在于，一开始它就把只有造物主才知道的真实值考虑了进去。如果我们知道了这个分布，则我们对于实验结果 $\mathbf{a}_{(0)}$ 的定量不确定度将无所不知。




真实参数 $\mathbf{a}_{true}$ 通过一组数据得以实现，即从这组数据拟合（观察）出参数 $\mathbf{a}_{0}$。如果实验重复多次，那么将得到新的数据集和新的拟合参数值。

图15.6.1　来自基本模型的数据集的统计全域

因此，我们工作的目的是，在不知道 $\mathbf{a}_{true}$ 以及不可能获得无穷多个假设的数据集时，寻找估计或近似 $\mathbf{a}_{(i)}-\mathbf{a}_{true}$ 的概率分布的方法。

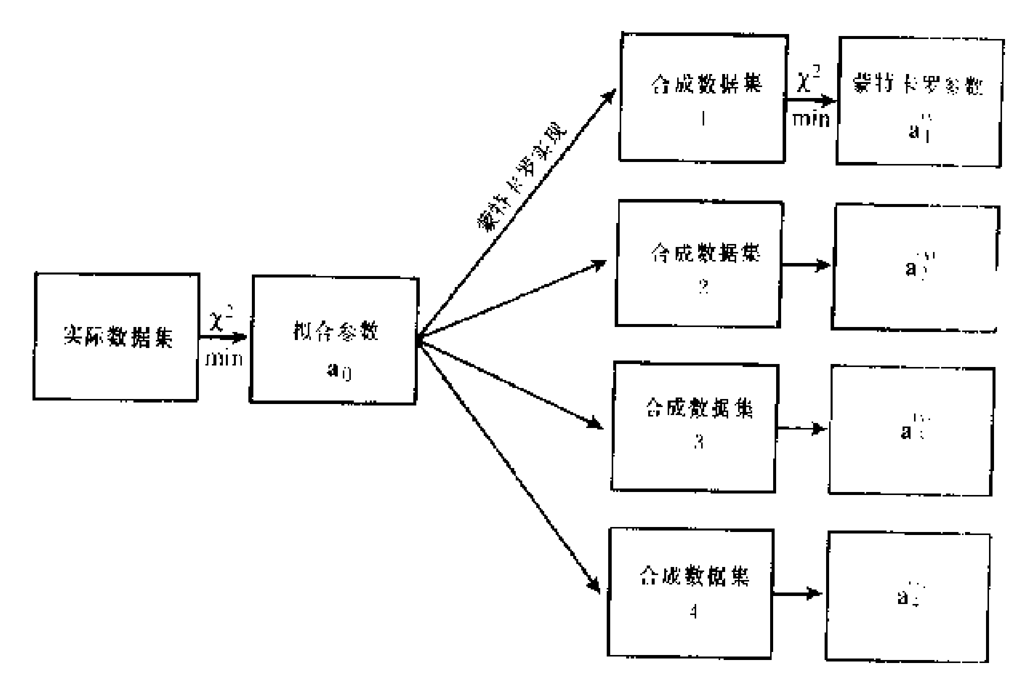
## 15.6.1　合成数据集的蒙特卡罗模拟

尽管测量的参数集 $\mathbf{a}_{(0)}$ 不是真实值，但我们可以设想一个虚构的空间，在这个空间里 $\mathbf{a}_{(0)}$ 是真实值。因为我们希望我们测量的参数值不要错误太大，所以我们又希望虚构空间和 $\mathbf{a}_{true}$ 所在的实际空间不相差太远。特别是，我们希望——不，还是让我们假设——在虚构空间中 $\mathbf{a}_{(i)}-\mathbf{a}_{true}$ 的概率分布的形状和真实空间中 $\mathbf{a}_{(i)}-\mathbf{a}_{true}$ 的概率分布的形状相同或者接近相同。注意我们没有假设 $\mathbf{a}_{(0)}$ 和 $\mathbf{a}_{true}$ 相等；他们肯定不相等。我们只假设引入实验和数据分析的，作为 $\mathbf{a}_{true}$ 函数的随机误差变化不快，因此 $\mathbf{a}_{(0)}$ 可作为一个合理的代替者。

现在在虚构空间里我们完全有能力计算 $\mathbf{a}_{(i)}-\mathbf{a}_{(0)}$ 的分布（参阅图15.6.2）。当给定一组假设的参数集 $\mathbf{a}_{(0)}$，如果我们对从它产生我们的数据的过程有所了解，则我们通常可以将这些参数的“综合”实现模拟成“合成数据集”。这个过程就是，从合适的分布中取得随机数（参阅第7.2节～第7.3节），以便模拟我们对于基本过程和仪器中测量误差的最正确理解。通过产生这些随机数，我们构造了一些数据集，这些数据集具有和实测数据点的个数相等，以及和所有控制（独立）变量的值精确地相同，如同我们实际的数据集的 $\mathscr{D}_{(0)}$ 一样。让我们称这些模拟的数据集为 $\mathscr{D}^S_{(1)}$，$\mathscr{D}^S_{(2)}$，……。根据构造，这些数据集和 $\mathbf{a}_{(0)}$ 的统计关系应该恰好与数据集 $\mathscr{D}_{(i)}$ 和 $\mathbf{a}_{true}$ 间具有的关系相同。（若读者还不知道怎样对正在测量的数据才能做一个可信的模拟的话，请阅读下面。）

其次，对每个 $\mathcal{D}^S_{(j)}$ 精确地执行同样的参数估算过程，即 $\chi^2$ 最小值法，它与为得到参数 $\mathbf{a}_{(0)}$ 而对实际数据执行的过程完全相同，由此得到模拟的测量参数 $\mathbf{a}^S_{(1)}, \mathbf{a}^S_{(2)}, \cdots$。每个模拟的测量参数集产生一个点 $\mathbf{a}^S_{(i)} - \mathbf{a}_{(0)}$。模拟足够多的数据集和寻出足够多的模拟被测参数，然后，就可以在 $M$ 维空间中画出所期望的概率分布。

事实上，人们以这种方式进行蒙特卡罗模拟的能力已经使当代实验科学的许多领域发生了革命。人们不仅仅能够用一种非常精确的方法来表征参数估算的误差的特征；而且还能够在计算机上试验出不同的参数估算方法，或者试验出不同的数据约化技术，以及根据任何期望的准则进行探索使结果的不确定性达到极小。如果要求作选择，是选择精通五英尺高的一书架解析统计学著作呢，还是选择掌握实施蒙特卡罗模拟的中等技能，当然我们毫无疑问地会选择具有后一种技能。




从实际实验得到的拟合参数作为真实参数的代替物。计算机生成的随机数用于模拟许多合成数据集。对每个这种数据集进行分析获得它的拟合参数。于是，对这些拟合参数围绕（已知）真实参数替代物的分布进行研究。

图15.6.2　实验的蒙特卡罗模拟

## 15.6.2　快速粗糙的蒙特卡罗方法：靴带法

当对基础的处理过程知之不多，或者对测量误差的性质不了解，以致不能进行合理的蒙特卡罗模拟时，本节提供了一个强有力的技术。假设，数据组由 N 个独立且完全相同分布的"数据点"组成。每个数据点可能由几个数组成，例如一个或多个控制变量（在确定要测量的范围内均匀分布）以及相应的测量值。独立且完全相同的分布意味着数据点的先后顺序不会影响获得拟合参数 a 的处理。例如，式(15.5.5)即 $\chi^2$ 的和式中，数据点被相加的顺序是无关的。最简单的例子是测试量的平均值或者测试量的某些函数的平均值。

靴带法(bootstrap)[1] 用 $N$ 个数据点组成的数据集 $\mathcal{D}^S_{(0)}$ 产生许多个合成数据集 $\mathcal{D}^S_{(1)}, \mathcal{D}^S_{(2)}, \ldots$，每个数据集也有 $N$ 个数据点。这个过程就是简单地用替换法从 $\mathcal{D}_{(0)}$ 中抽出 $N$

数据点。因为是替换，所以不必每次回到原始数据点。在得到的数据集中，原始数据的随机的一部分，典型的是 $\sim\frac{1}{e}\approx37\%$ 被复制的点所替代。现在，和前面讨论的完全一样，对这些数据点进行对实际数据处理完全相同的估计过程，得到一组模拟的测量参数 $\mathbf{a}^{S}_{(1)},\mathbf{a}^{S}_{(2)},\cdots$，它们围绕 $\mathbf{a}_{(0)}$ 的分布和 $\mathbf{a}_{(0)}$ 围绕 $\mathbf{a}_{true}$ 的分布非常接近。

看起来似乎什么也没有得到，是不是？实际上，为了使靴带法被统计学家们接受，花了十年多的时间。到现在，很多定理的证明说明了靴带法的可靠性(看参考文献[2])。在靴带法中隐含的基本思想是，实际数据组在被测量的数值处看做是由 $\delta$ 函数组成的概率分布。在大多数情况下，这是最好的甚至是唯一可能的对基本概率分布的估计。做到这一点需要勇气，但是人们通常简单地将那种分布作为蒙特卡罗模拟的基础。

注意，当靴带法的这种独立完全相同分布的假设不满足的情况。例如，对一些控制变量在空间等间隔上进行测量，那么通常可以排除这些点是独立完全等同分布的假设，而它们是在测量的范围内均匀分布。但是，$\mathbf{a}$ 的某些估计(例如有关傅里叶方法)可能对出现在网格上的点特别敏感。在那种情况下，靴带法将给出错误的分布。也还必须注意，当估计量看做是在 $N$ 个数据点中的小尺度量的块，或者估计量排序数据并看成是相继的差时，显然靴带法也不适用于这些情况。(证明这些方法的公理还是正确的，但是其中的一些技巧性假设被这些例子破坏了。)

然而，在大多数情况下，靴带法能容易而又非常迅速地产生在估计参数集中的误差的蒙特卡罗估算。

### 15.6.3 置信界限

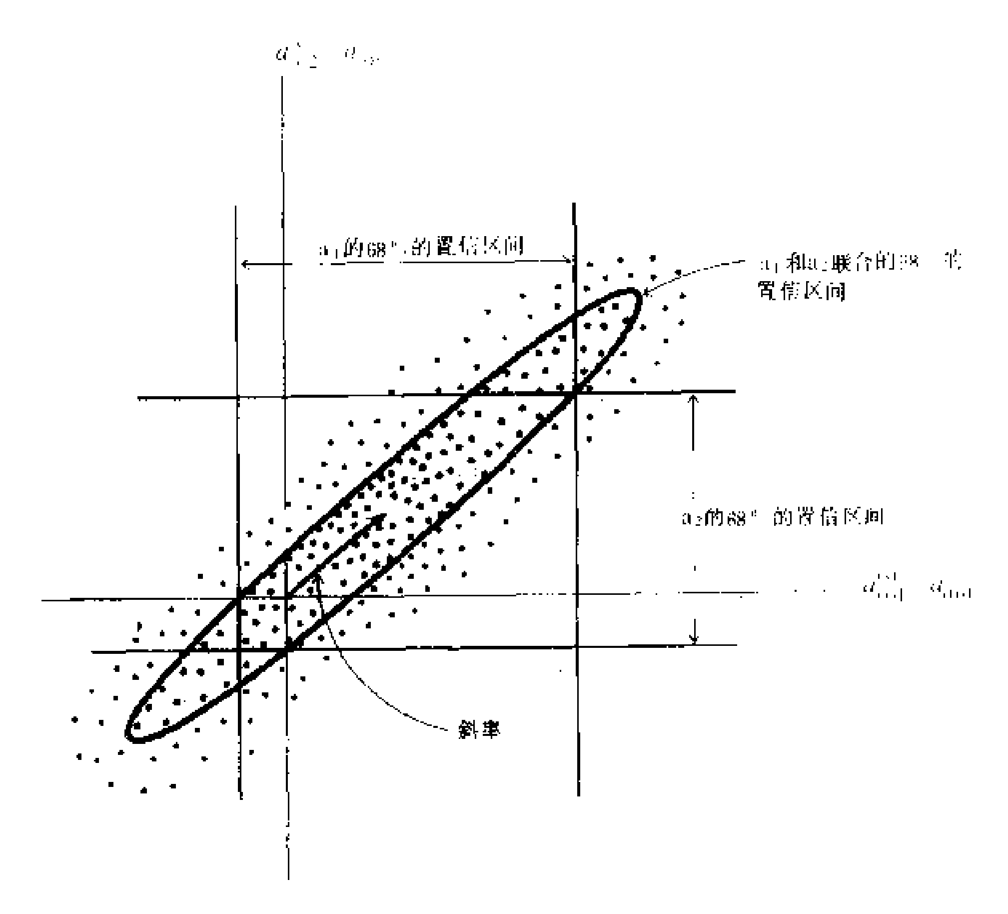
为了掌握分布的概貌，我们通常是按**置信界限**的形式，而不是对参数估算误差的概率分布提供所有的细节。整个概率分布是定义在参数 $\mathbf{a}$ 的 $M$ 维空间上的一个函数，一个**置信区域**(或者**置信区间**)恰好是 $M$ 维空间的一个区域(希望是一个小的区域)，这个区域包含全部概率分布的一定百分比(通常希望此值大)。例如可以指着一个置信区域说："真实参数有99% 的机会落入围绕被测值的这个区域内。"

值得强调的是，作为一个实验者，必须同时拾取**置信水平**(在上面的例子中为 99%)及置信区域的形状。唯一的要求是，置信区域必须包括所陈述的概率百分比。一些百分比是科学中惯用的，它们是：68.3%(可引用的最低置信水平)，90%、95.4%、99%和 99.73%。更高的置信水平可方便地写为" 99.9……9 "。至于置信区域的形状，显然想要的是，一个合理地围绕所测量值 $\mathbf{a}_{(0)}$ 为中心的紧凑的区域，因为找一个置信界限的全部目的在于确定被测值的可信度，在一维问题中，置信区域是一个以测量值为中心的线段；而在高维问题中，经常用到椭圆或椭球。

读者可能怀疑数字 68.3%、95.4%、99.73% 以及椭球的应用可能和正态分布有联系。从历史的角度看这是完全正确的，但是今天并不总是和正态分布有联系。一般说来，参数概率分布不是正态分布，前面所用作为置信水平的数，纯粹是出于惯例。

图15.6.3画出了在 $M=2$ 的情形下一个可能的概率分布图。该图显示了三个可能有用的不同的置信区域，它们都有相同的置信水平。两条垂直线围住的一条窄带(水平区间)，它代表在不考虑 $a_2$ 值时变量 $a_1$ 的 68% 的置信区间。同样，两条水平线为 $a_2$ 围住了 68% 置信

区间。椭圆则代表了 $a_1$ 和 $a_2$ 联合的 68% 的置信区间。注意，为了覆盖和两条带形相同的概率，椭圆必须扩展到两条带形之外(在下面我们还要返回来讨论这一点)。




测量数据点的相同部分(这里是68%)位于(i)在两条垂直线间;(ii)在两条水平线之间;(iii)在椭圆内.

图15.6.3 1维和2维的置信区间

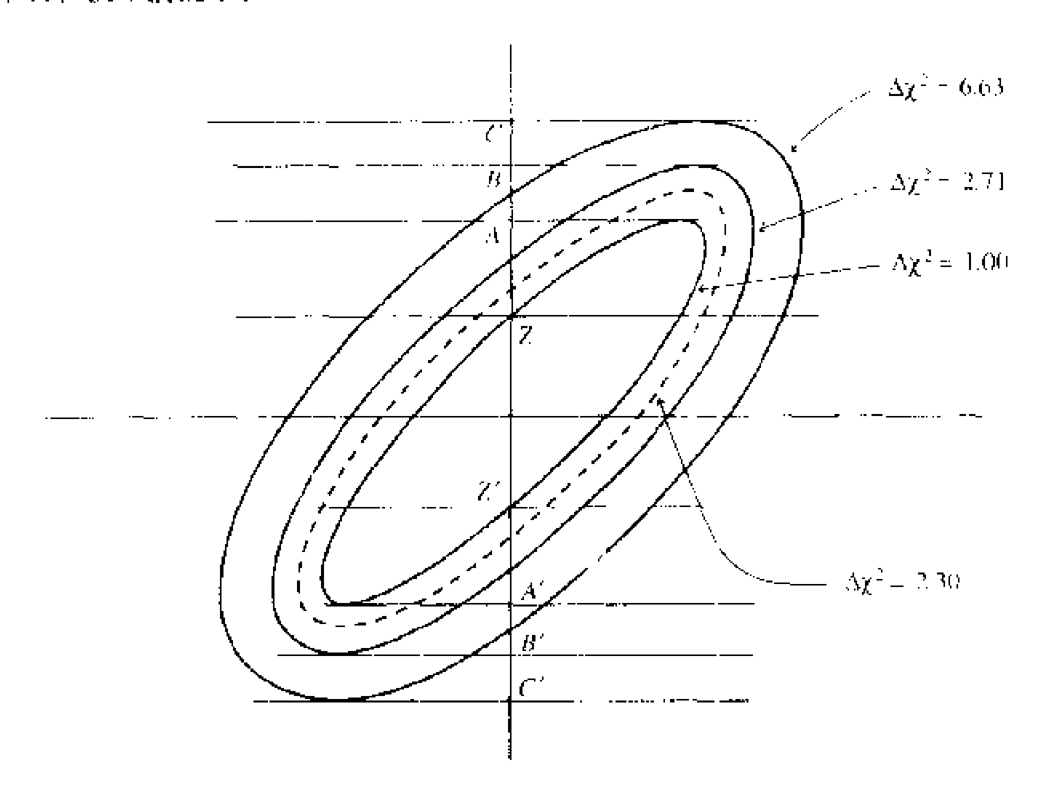
## 15.6.4 常数 chi 平方边界作为置信界限

当用于估计参数 $\mathbf{a}_{(0)}$ 的方法是 chi 平方最小值方法，即我们在这一章前几节讨论的，则对于置信区间形状有一个自然的选择，它的应用几乎是普适的。对于观测到的数据集 $\mathcal{D}_{(0)}$ 而言，$\chi^2$ 在 $\mathbf{a}_{(0)}$ 处取极小值。称这个极小值为 $\chi^2_{min}$。如果参数向量 $\mathbf{a}$ 的值偏离 $\mathbf{a}_{(0)}$，则 $\chi^2$ 将增加。使 $\chi^2$ 增加值小于 $\Delta\chi^2$ 的区域可定义为，围绕 $\mathbf{a}_{(0)}$ 的 $M$ 维空间的置信区域。如果 $\Delta\chi^2$ 是一个大的数值，则这个区域也较大；如果 $\Delta\chi^2$ 比较小，则这个区域也较小。如果想要这个区域包括上面所定义的 $\mathbf{a}$ 的概率分布的某一部分，例如 68%，90% 等，则必须选择不同的 $\Delta\chi^2$。这些区域可看作参数 $\mathbf{a}_{(0)}$ 的置信区域。

人们常常对全部 $M$ 维置信区域不感兴趣，而对某些较少的 $\nu$ 个参数的单独置信区域感兴趣。例如，人们可能对每个参数的单独的置信区间感兴趣(如在图15.5.3中的带形)，在那种情况下 $\nu=1$。基于这种考虑，$M$ 维参数空间中的 $\nu$ 维子参数空间的置信区域，是在固定 $\Delta\chi^2$ 定义下的 $M$ 维区域在我们感兴趣的 $\nu$ 维空间中的投影。在图15.6.4的情况下 $M=2$，我们标出了不同 $\Delta\chi^2$ 值对应的区域。由 $\Delta\chi^2=1$ 限定的区域，所对应的变量 $a_2$ 的一维置信区间位

于直线 $A$ 和 $A'$ 之间。

注意，采用的是高维区域在低维空间中的投影，而不是采用它们的相交。在图15.6.4中，相交部分是 $Z$ 和 $Z'$ 之间的带形，它永远不会被用到。在图中加以显示的目的只是为了提醒读者不要和投影搞混了。




将 $\Delta\chi^2=1.00$、2.71、6.63 所对应的实线椭圆投影到一维轴上得到区间 $AA'$，$BB'$，$CC'$。这些区间——不是椭圆本身——包含了 68.3%、90% 和 99% 的正态分布数据点。包含 68.3% 的正态分布数据点的椭圆用虚线表示，它对应的 $\Delta\chi^2$ 是 2.30。对其它的数据，请看相关的表格。

图15.6.4　$\chi^2$ 值大于拟合极小值所对应的椭圆置信区域

## 15.6.5　正态情况下参数的概率分布

读者可能要问为什么迄今为止，我们的讨论还和 $\chi^2$ 拟合过程的误差估计，最注目的协方差矩阵 $C_{ij}$，没发生任何联系。原因在于：$\chi^2$ 极小化方法是一个非常有用的参数估计方法，即使测量误差不是正态分布也适用。尽管如果 $\chi^2$ 参数估计成为最大似然估计（第15.1节），但要求误差是正态分布时，人们还是放弃最大似然估计的这个特性，而运用相对方便的 $\chi^2$ 处理。只有在极端的情况下，测量误差分布带有很大的"尾巴"时，人们才放弃 $\chi^2$ 极小化方法，而采用更加稳健的技术，我们将在第15.7节讨论这一方法。

不过，只有在测量误差实际上呈正态分布（或者达到这种程度）时，由 $\chi^2$ 极小化方法导出的形式协方差矩阵才有意义。在误差分布不是正态时，可"允许"

- 通过求 $\chi^2$ 极小值获得拟合参数。
- 用常数 $\Delta\chi^2$ 的一个轮廓线作为置信区域的边界线。
- 用蒙特卡罗模拟方法或者详细的解析计算确定哪一条 $\Delta\chi^2$ 轮廓能正确地表示了所想

要的置信水平。

- 把协方差矩阵 $C_j$ 作为"拟合的形式协方差矩阵"。

但不允许

- 使用在正态误差情况下给出的公式，这些公式表示 $\Delta\chi^2$、$C_j$ 以及置信水平之间的定量关系。

这里列举在以下条件下成立的关键定理：(i)测量误差是正态分布；以及要么(ii)模型对于它的参数是线性的；要么(iii)样本数目足够大，拟合参数 **a** 的不确定度没有扩展到区域以外，在此区域中模型可用一个合适的线性模型来代替。[注意条件(iii)并没有排除使用一个非线性拟合程序例如 **mqrfit** 来寻找拟合参数值。]

**定理 A** $\chi^2_{min}$的分布是自由度为 $N-M$ 的 $\chi^2$ 分布，这里 $N$ 是数据点的个数，$M$ 是拟合参数的个数。这是一个估算模型拟合优度的基本定理，如同我们在第15.1节中讨论的那样。我们首先将这条定理列出来是为了提醒读者，除非拟合优度是可信的，否则整个参数估计过程是不可靠的。

**定理 B** 如果 $\mathbf{a}_j$ 是从实际参数 $\mathbf{a}_{(0)}$ 对应的模拟数据的全域中导出的，则 $\delta\mathbf{a}\equiv\mathbf{a}_{(j)}-\mathbf{a}_{(0)}$ 的概率分布是多元正态分布

$$P(\delta\mathbf{a})\,da_1\ldots da_M=\text{常数}\times\exp\left(-\frac{1}{2}\delta\mathbf{a}\cdot[\alpha]\cdot\delta\mathbf{a}\right)da_1\ldots da_M$$

其中，$[\alpha]$是在等式(15.5.8)中定义的曲率矩阵。

**定理 C** 如果 $\mathbf{a}^S_{(j)}$ 是从实际参数 $\mathbf{a}_{(0)}$ 对应模拟数据集的全域中导出的，则量 $\Delta\chi^2\equiv\chi^2(\mathbf{a}_{(j)})-\chi^2(\mathbf{a}_{(0)})$的分布呈现为自由度为 $M$ 的 $\chi^2$分布。在这里，$\chi^2$值都用固定(实际)的数据集 $\mathscr{D}_{(0)}$ 估算。这个定理把特定的 $\Delta\chi^2$值和它所包围的 $M$ 维区域对应的概率分布部分，也即 $M$ 维置信区域的置信水平之间建立了联系。

**定理 D** 假设 $\mathbf{a}^S_j$ 是从模拟数据集的全域中导出的(如同上面一样)，还假设它的前 $\nu$ 个分量 $a_1,\ldots,a_\nu$ 保持固定值，剩下的 $M-\nu$ 个分量使 $\chi^2$取极小值。称这个极小值为 $\chi^2_\nu$，则 $\Delta\chi^2_\nu\equiv\chi^2_\nu-\chi^2_{min}$是自由度为 $\nu$ 的 $\chi^2$分布。如果查看图15.6.4，就会明白这个定理将 $\Delta\chi^2$的**投影**区域和置信水平之间联系了起来，在图中，让 $a_2$固定，变化 $a_1$，使 $\chi^2$取极小值的点描出一个椭圆，椭圆的顶部和底部分别和常数 $a_2$对应直线相切，因此椭圆也就是能将它投影到较低维空间的线。

作为第一个例子，让我们考虑 $\nu=1$ 的情况，其中我们要找某单变量(比如 $a_1$)的置信区间。注意自由度 $\nu=1$ 的 $\chi^2$分布和单个正态分布量的平方分布相等。因此 $\Delta\chi^2_\nu<1$ 发生的概率为 68.3%(对于正态分布为 $1-\sigma$)，$\Delta\chi^2_\nu<4$ 发生的概率为 95.4% (对于正态分布为 $2-\sigma$)，$\Delta\chi^2_\nu<9$ 发生的概率为 99.73%(对于正态分布即 $3-\sigma$)。用这种方法就能找到和所想要的置信水平对应的 $\Delta\chi^2_\nu$(其它的数据在表15.6.5.1中给出)。

**表15.6.5.1　$\Delta\chi^2$作为置信水平和自由度的函数**

| $p$ | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 68.3% | 1.00 | 2.30 | 3.53 | 4.72 | 5.89 | 7.04 |
| 90% | 2.71 | 4.61 | 6.25 | 7.78 | 9.24 | 10.6 |
| 95.4% | 4.00 | 6.17 | 8.02 | 9.70 | 11.3 | 12.8 |
| 99% | 6.63 | 9.21 | 11.3 | 13.3 | 15.1 | 16.8 |
| 99.73% | 9.00 | 11.8 | 14.2 | 16.3 | 18.2 | 20.1 |
| 99.99% | 15.1 | 18.4 | 21.1 | 23.5 | 25.7 | 27.8 |

设 $\delta\mathbf{a}$ 表示参数变化，它的第一个分量是任意的 $\delta a_1$，但是其余分量选择将使 $\Delta\chi^2$ 取极小值。于是可以应用定理 D。$\Delta\chi^2$ 值通常由下式给出。

$$\Delta\chi^2 = \delta\mathbf{a} \cdot [\alpha] \cdot \delta\mathbf{a} \tag{15.6.1}$$

它由将等式(15.5.8)用到 $\chi^2_{\min}$ 处而得出，其中 $\beta_k = 0$。因为根据假设，$\delta\mathbf{a}$ 除了第一个分量外，其余分量使 $\chi^2$ 取最小值，所以正规方程(15.5.9)中第二到第 $M$ 个方程依然有效，因此，式(15.5.9)的解是

$$\delta\mathbf{a} = [\alpha]^{-1} \cdot \begin{pmatrix} c \\ 0 \\ \cdot \\ \cdot \\ \cdot \\ 0 \end{pmatrix} = [C] \cdot \begin{pmatrix} c \\ 0 \\ \cdot \\ \cdot \\ \cdot \\ 0 \end{pmatrix} \tag{15.6.2}$$

其中 $c$ 是一个任意的常数，我们可以调整 $c$，使式(15.6.1)给出期望的的左端值。将式(15.6.2)代入式(15.6.1)，并且注意到$[C]$和$[\alpha]$互为逆矩阵，我们得到

$$c = \delta a_1/C_{11} \quad \text{和} \quad \Delta\chi^2_\nu = (\delta a_1)^2/C_{11} \tag{15.6.3}$$

或者

$$\delta a_1 = \pm\sqrt{\Delta\chi^2_\nu}\sqrt{C_{11}} \tag{15.6.4}$$

注意最后一个等式！它给出了置信区间 $\pm\delta a_1$ 和形式标准差 $\sigma_1 \equiv C_{11}$ 之间的关系。毫无疑问，我们发现 68% 的置信区间是 $\pm\sigma_1$，95% 的置信区间是 $\pm 2\sigma_1$ 等等。

以上的讨论不只对单个参量 $a_i$ 成立，而且对它们的任何线性组合都成立：如果

$$b \equiv \sum_{k=1}^{M} c_i a_i = \mathbf{c} \cdot \mathbf{a} \tag{15.6.5}$$

则 $b$ 的 68% 置信区间是

$$\delta b = \pm\sqrt{\mathbf{c} \cdot [C] \cdot \mathbf{c}} \tag{15.6.6}$$

但是，这种简单的、如同正态分布的数值关系在 $\nu>1$ 的情况下不成立。特别是在 $\nu>1$ 时，$\Delta\chi^2=1$ 并不是 68.3% 的置信区域，也不是它投影到边界上来。如果要计算不是单个参量的置信区间，而是两个参量的联合置信椭圆，或者三个参量的置信椭球，或者更高，那么必须按照下面的步骤实施上面所说的定理 C 和 D。

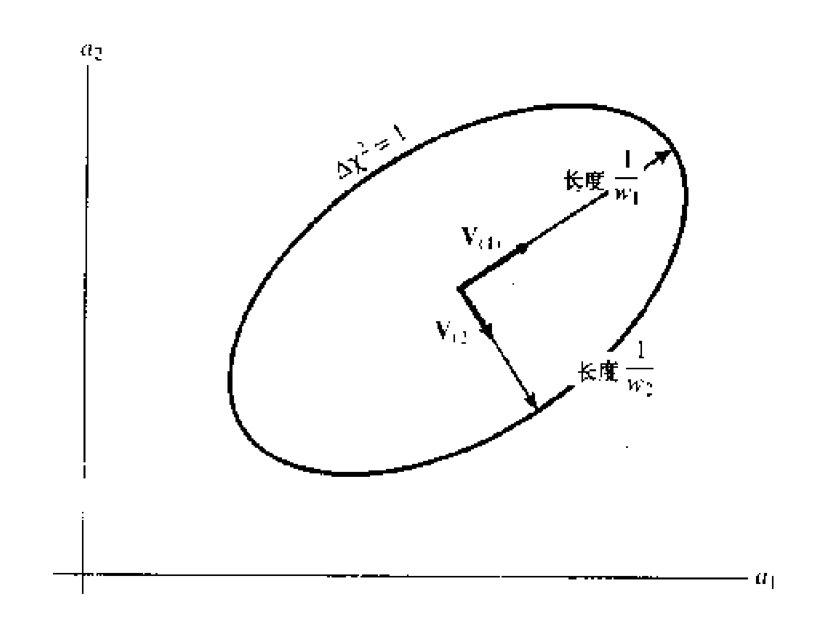
- 令 $\nu$ 为所想显示的联合置信区域的拟合参数数目，$\nu \leqslant M$，称这些参数为“感兴趣参数”。
- 令 $p$ 为所期待的置信限，例如 $p=0.68$ 或 $p=0.95$。

- 寻找$\Delta$(也就是$\Delta\chi^2$)使得一个自由度为$\nu$的$\chi^2$的变量小于$\Delta$的概率是$p$。对某些$p$和$\nu$的有用值,$\Delta$用表格给出。对于其它的值,可以利用程序**gammp**和一个简单的求根程序(例如二分法)寻找$\Delta$使**gammq**$(\nu/2,\Delta/2)=1-p$。
- 将chi平方拟合所得的$M\times M$协方差矩阵求逆,即$[C]=[\alpha]^{-1}$,将所感兴趣的参数对应的$\nu$行和$\nu$列相交处的元素重写到一个$\nu\times\nu$的矩阵,命名为$[C_{proj}]$。
- 对矩阵$[C_{proj}]$求逆。(在一维情况下,只需将元素$C_{11}$求倒数即可)。
- 在所感兴趣的$\nu$维子空间中,所期望的置信区域的椭圆边界方程是

$$\Delta = \delta\mathbf{a}'[C_{proj}]^{-1}\cdot\delta\mathbf{a}' \qquad (15.6.7)$$

其中$\delta\mathbf{a}'$是感兴趣参数的$\nu$维向量。

如果对这一点还有疑惑,那么将图15.6.4和附加的表格进行比较会大有帮助。读者一定能证实下列论述:考虑$M=2$及$\nu=1$和$\nu=2$的情况,(i)在$C$和$C'$之间的水平带形包含了99%的概率分布,因此也就是$a_2$单独置信水平下的置信界限;(ii)和上面类似,在$B$和$B'$之间的水平带形是置信水平为90%的置信界限;(iii)虚线的椭圆,以$\Delta\chi^2=2.30$标出,包含了68.3%的概率分布,因而它是在此置信水平下,关于$a_1$和$a_2$的联合置信区域。




向量$\mathbf{V}_{(i)}$是沿着置信区域主轴的单位向量。半轴的长度等于奇异值$w_i$的倒数。如果轴用某个常数因子$\alpha$来标度的话,那么,$\Delta\chi^2$将用因子$\alpha^2$来标度。

图15.6.5 $\Delta\chi^2=1$的置信区域椭球和由奇异值分解计算的量之间的关系

### 15.6.6 奇异值分解下的置信界限

如果是用奇异值分解法(第15.4节)求$\chi^2$拟合,那么拟合的形式误差的信息将以不同的形式出现,但通常是更加方便的形式。矩阵$\mathbf{V}$的各列是正交的$M$个向量,它们分别是$\Delta\chi^2=$常数的椭球的主轴。我们将每一列分别表示为$\mathbf{V}_{(1)}\cdots\mathbf{V}_{(M)}$。这些主轴的长度和对应的奇异值$w_1\cdots w_M$成反比,参看图15.6.5。椭球的边界由下式给出:

$$\Delta\chi^2 = w_1^2(\mathbf{V}_{(1)}\cdot\delta\mathbf{a})^2 + \ldots + w_M^2(\mathbf{V}_{(M)}\cdot\delta\mathbf{a})^2 \qquad (15.6.8)$$

上式是前面式(15.4.18)的改写形式。记住,在给出了一列椭球的向量主轴情况下,画椭球的图形,比给出它的二次矩阵形式后画椭球要容易得多。

协方差矩阵[C]的公式用向量 $\mathbf{V}_{(i)}$ 来表示将成为

$$[C] = \sum_{i=1}^{M} \frac{1}{w_i^2} \mathbf{V}_{(i)} \otimes \mathbf{V}_{(i)} \tag{15.6.9}$$

或者用分量

$$C_{jk} = \sum_{i=1}^{m} \frac{1}{w_i^2} V_{ji} V_{ki} \tag{15.6.10}$$


参考文献和进一步读物:

Efron, B. 1982, *The Jackknife, the Bootstrap, and Other Resampling Plans* (Philadelphia: S. I. A. M.). [1]

Efron, B., and Tibshirani, R. 1986, *Statistical Science* vol. 1, pp. 54~77. [2]

Avni, Y. 1976, *Astrophysical Journal*, vol. 210, pp. 642~646. [3]


## 15.7 稳健估计

**稳健**的概念在前面已经提到过几次。在第14.1节中我们注意到,中位数是比平均值对中心值的一个更稳健的估计,在第14.6节中曾提到秩相关比线性相关更加稳健。对于因实验误差而引起的偏离点不能用高斯模型的情况,我们也已在第15.1节中进行了讨论。

稳健的概念是1953年由 G. E. P. Box 在统计学中提出的。对此术语具有各种不同的严格的或不太严格的数学定义,但是一般说来,稳健是一个统计估计量,它意味着"对最佳估计量所假设的理想模型,发生微小偏离的不敏感性"。"微小"的意义有两种不同的解释,都是很重要的:或者是所有的数据点都有一个小的偏离,或者是数据点中很少的几个点存在较大的偏离。对于后一情况,引起了偏离点的概念,这在统计过程中通常是最强调的。

统计学家们已经发展了各种不同的稳健估计统计方法。它们中的很多(即使不是全部)可以分为三类。

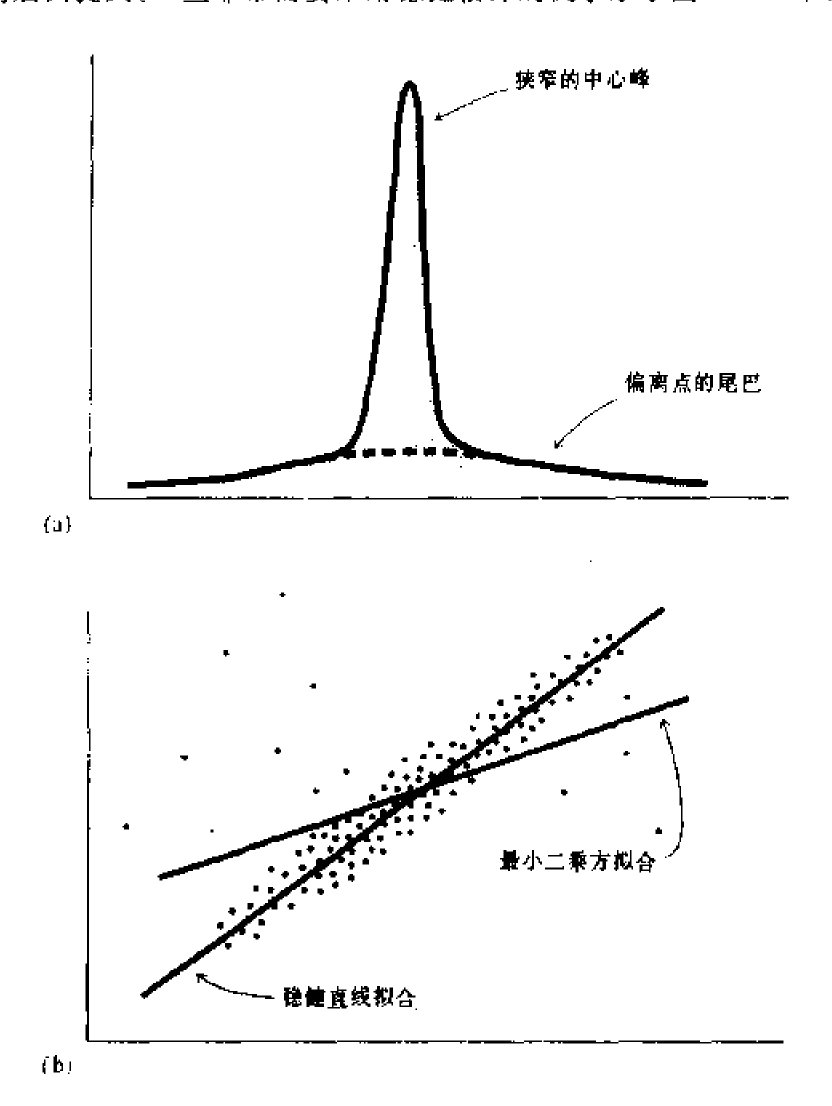
**M 估计**是建立在最大似然方法上的,类似于由等式(15.1.3)导出的等式(15.1.5)和(15.1.7)。$M$ 估计通常是和模型拟合,也就是参数的估计,关系最密切的一类,因此在下面我们将详细地讨论这类估计。

**L 估计**是"顺序统计量的线性组合"。这类估计方法最适用于中心值和集中趋势的估计,它们偶尔也用于某些参数估计问题。两种"典型"的 $L$ 估计方法将给出一个一般的概念,它们是(i)中位数;和(ii)塔凯(Tukey)的三重均值,它定义为某个分布的第一、第二和第三个四分位点的加权平均,权重分别为 1/4、1/2 和 1/4。

**R 估计**是建立在秩检验上的估计,例如两个分布的相等或不等可以用**威尔科克斯**(Wilcoxon)检验来估计,这种估计是在两个分布的组合样本中计算一个分布平均秩。柯尔莫哥洛夫-史密诺夫(Kolmogorov-Smirnov)统计(等式(14.3.6))和斯皮尔曼(Spearman)秩阶相关系数(等式(14.6.1),本质上都是 $R$ 估计,尽管它们的定义的形式上不是 $R$ 估计。

其它的一些稳健估计方法,出自最优控制和最优滤波领域而不是数理统计领域,我们将

在这一节的后面提到。一些非常需要采用稳健估计的例子示于图15.7.1中。




(a)一个带有偏离点尾巴的一维分布；在这些偏离点处的统计起伏可以阻止精确地确定中央峰的位置。(b)拟合成一条直线的二维分布；非稳健估计技术例如最小二乘方拟合具有对偏离点产生不期望的敏感性。

图15.7.1 稳健统计方法最适用的两个例子

## 15.7.1 用局部 $M$ 估计法估计参数

假设我们知道我们的测量误差不服从正态分布，那么在导出模型 $y(x;\mathbf{a})$ 的估计参数 $\mathbf{a}$ 的最大似然等式时，我们将用下式代替等式(15.1.3)：

$$P=\prod_{i=1}^{N}\{\exp[-\rho(y_i,y\{x_i;\mathbf{a}\})]\Delta y\} \qquad (15.7.1)$$

其中函数 $\rho$ 是概率密度的负对数。和式(15.1.4)类似，对式(15.7.1)取对数，我们发现我们要极小化下式

$$\sum_{i=1}^{N}\rho(y_i,y\{x_i;\mathbf{a}\}) \qquad (15.7.2)$$

常常会出现这种情况，即函数 $\rho$ 并不是完全依赖于它的两个自变量(测量的 $y_i$ 和预测的 $y(x_i)$)而是只依赖于它们的差，至少在用我们分配到各点的权重因子 $\sigma_i$ 来量度时就是如此。在这种情况下，$M$ 估计称为是局部的，我们可以用下面的表述代替式(15.7.2)：

$$\text{对 } \mathbf{a} \text{ 求 } \sum_{i=1}^{N} \rho\left(\frac{y_i - y(x_i;\mathbf{a})}{\sigma_i}\right) \quad \text{的极小化} \tag{15.7.3}$$

其中函数 $\rho(z)$ 的单变量 $z \equiv [y_i - y(x_i)]/\sigma_i$ 的函数。

如果我们定义 $\rho(z)$ 的微分函数为 $\psi(z)$，

$$\psi(z) \equiv \frac{d\rho(z)}{dz} \tag{15.7.4}$$

则推广式(15.1.7)，就得一般的 $M$ 估计情形下的等式

$$0 = \sum_{i=1}^{N} \frac{1}{\sigma_i}\psi\left(\frac{y_i - y(x_i)}{\sigma_i}\right)\left(\frac{\partial y(x_i;\mathbf{a})}{\partial a_k}\right) \quad k = 1,\dots,M \tag{15.7.5}$$

如果将式(15.7.3)和(15.1.3)，(15.7.5)和(15.1.7)比较，就会发现在误差呈正态分布的特殊情形下有

$$\rho(z) = \frac{1}{2}z^2 \qquad \psi(z) = z \quad (\text{正态}) \tag{15.7.6}$$

如果误差呈**二重或双边指数函数**分布，也就是

$$\text{Prob}\,\{y_i - y(x_i)\} \sim \exp\left(-\left|\frac{y_i - y(x_i)}{\sigma_i}\right|\right) \tag{15.7.7}$$

则对照之下，

$$\rho(z) = |z| \qquad \psi(z) = \text{sgn}(z) \quad (\text{二重指数}) \tag{15.7.8}$$

比较等式(15.7.3)，我们发现在这种情形下，最大似然估计量是通过求**平均绝对偏差**极小来获得，而不是对均方偏差取极小来得到的。这里，尽管分布尾巴仍呈指数衰减，但是它的渐近结果比相应的高斯函数大很多。

带有更广泛的——因此有时也更加真实的——尾部的分布是柯西(Kauchy)分布或者洛伦兹(Lorentzian)分布

$$\text{Prob}\,\{y_i - y(x_i)\} \sim \frac{1}{1 + \frac{1}{2}\left(\frac{y_i - y(x_i)}{\sigma_i}\right)^2} \tag{15.7.9}$$

这隐含着

$$\rho(z) = \log\left(1 + \frac{1}{2}z^2\right) \qquad \psi(z) = \frac{z}{1 + \frac{1}{2}z^2} \quad (\text{洛伦兹}) \tag{15.7.10}$$

注意，在推广的正规方程组(15.7.5)中，函数 $\psi$ 作为一个权重函数而出现。当误差呈正态分布时，等式(15.7.6)说明数据点的偏离值越大，权重越大。与此不同的是，当分布的尾部稍微突出些，如式(15.7.7)，则式(15.7.8)说明所有的偏离点得到相同的相对权重，而只用到了它们的符号信息。最后，如果尾部更大，则式(15.7.10)说明随着偏离值的增大，$\psi$ 开始增加，然后开始减小，以致非常大的偏离点——真正的偏离点——在参数的估值中根本没有计算在内。

由个别数据点给出的权重随着偏离值的增加开始递增，然后递减的一般概念促使人们

对 $\psi$ 进行一些附加的规定，它和标准的教科书上的概率分布并不对应。下面有两个例子：

**安德鲁正弦函数**(Andrew's sine)

$$\psi(z) = \begin{cases} \sin(z/c) & |z| < c\pi \\ 0 & |z| > c\pi \end{cases} \tag{15.7.11}$$

如果测量误差恰好是正态分布，标准差为 $\sigma_i$，则可以证明常数 $c$ 的最优值是 $c = 2.1$。

**塔凯双权**(Takey's biweight)

$$\psi(z) = \begin{cases} z(1 - z^2/c^2)^2 & |z| < c \\ 0 & |z| > c \end{cases} \tag{15.7.12}$$

对于正态分布误差，上式中 $c$ 的最优值是 $c = 6.0$。

### 15.7.2　$M$ 估计的数值计算

用 $M$ 估计的方法拟合一个模型，首先必须确定想要哪一种 $M$ 估计方法，也就是想要用哪一对匹配的 $\rho$、$\psi$。我们倾向于使用式(15.7.8)或者(15.7.10)。

接着，必须在两个一样难度的问题中作霍布森(Hobson)选择。要么解 $M$ 个方程构成的非线性方程组(15.7.5)，要么求含 $M$ 个变量的单一函数的极小化(15.7.3)。

注意，在式(15.7.8)的函数中，$\psi$ 是不连续的，$\rho$ 的导数也是不连续的。这些不连续性常常会在一般的非线性方程求解和求函数极小化的过程中引起大的混乱。现在可能想舍弃式(15.7.8)，而采用式(15.7.10)，因为式(15.7.10)中的函数比较光滑。但是，也将发现后一选择对很多方程组的求解或求极小化程序来说，有很多糟糕之处：拟合参数的微小变化可能使 $\psi(z)$ 偏离它的峰值，而进入它小的渐近值区域。因此，方程中的不同项有时起作用，有时不起作用(和解析不连续性几乎一样糟糕。)

但是不要绝望！如果读者的计算机(或者对于个人计算机来说，忍耐力)能胜任，则这对于向下单纯形极小化算法是一个非常好的应用事例。这一算法示于第10.4节的 **amoeba** 和第10.9节中的 **amebsa**。此算法对于连续性没有作任何假设，它只是体现向下滑的意图。它对函数 $\rho$ 的任何合理选择算法本质上都有效。

无论如何找一个好的初始值是非常有利的(更加经济)。通常，我们首先用标准的 $\chi^2$(非稳健统计)方法拟合模型，如我们在第15.4节或第15.5节中所叙述的那样。接着用所得的结果作为程序 **amoeba** 的初始值，并且按稳健统计方法挑选 $\rho$，再求(15.7.3)表达式的极小值。

### 15.7.3　通过极小化绝对偏差拟合直线

有时偶然会出现比上面所讨论的一般技巧更容易的情况，当模型是一条简单的直线

$$y(x;a,b) = a + bx \tag{15.7.13}$$

并且其中各点的权重 $\sigma_i$ 都相等时，等式(15.7.7)～(15.7.8)就是这种情况。这个问题恰好是在等式(15.2.1)中提到的要采用稳健估计方法的问题，也就是用一组数据点拟合一条直线。要求其极小化的优值函数是

$$\sum_{i=1}^{N} |y_i - a - bx_i| \tag{15.7.14}$$

而不是由式(15.2.2)给出的 $\chi^2$。

简化的关键是基于以下事实：一组数 $c_i$ 的中位数 $c_M$ 也就是使偏差绝对值的和达到极小

的值

$$\sum_i |c_i - c_M|$$

(证明:在上式中对 $c_M$ 求微分,并令它为 0)。

由此我们知道对固定的 $b$,使式(15.7.14)最小的 $a$ 为

$$a = \{y_i - bx_i\} \text{ 的中位数} \tag{15.7.15}$$

对于参数 $b$,方程(15.7.5)变为

$$0 = \sum_{i=1}^{N} x_i \operatorname{sgn}(y_i - a - bx_i) \tag{15.7.16}$$

(其中 sgn(0)看作为零)。如果我们把(15.5.15)中隐含的函数关系 $a(b)$ 代替上式中的 $a$,那么我们将得到一个含有单变量的方程,它可以采用我们在第9.1节叙述的二分法或划界法求解。(事实上,如果采用更高级的求根方法将非常危险,因为等式(15.7.16)中的函数是不连续的。)

下面是一个完成以上所述的程序。它调用第8.5节中的 **select** 求中位数。划界法和二分法都用于下面的程序中,用来计算 $\chi^2$的结果,此结果将产生 $a$ 和 $b$ 的初始猜测值。注意(15.7.16)的右端的计算是采用程序 **rofunc**,它通过全局(最高级别)变量和程序 **medfit** 进行数据传输。

```
#include <math.h>
int ndatat;
float *xt, *yt, aa, abdevt;

void medfit(float x[], float y[], int ndata, float *a, float *b, float *abdev)
    采用最小绝对偏差的准则拟合直线 y=a+bx。数组 x[1..ndata]和 y[1..ndata]是输入实验数据点。输出拟合参数 a
    和 b,同时输出的还有来自拟合直线的实验数据点(关于 y 的)平均绝对偏差 abdev。此程序还要用到程序 rofunc,它
    们之间通过全局变量进行数据交换。
{
    float rofunc(float b);
    int j;
    float bb,b1,b2,del,f,f1,f2,sigb,temp;
    float sx=0.0,sy=0.0,sxy=0.0,sxx=0.0,chisq=0.0;

    ndatat=ndata;
    xt=x;
    yt=y;
    for (j=1;j<=ndata;j++) {            对于 a 和 b 的初始值,我们将寻找最小二乘拟合直线
        sx += x[j];
        sy += y[j];
        sxy += x[j]*y[j];
        sxx += x[j]*x[j];
    }
    del=ndata*sxx-sx*sx;
    aa=(sxx*sy-sx*sxy)/del;             最小二乘解
    bb=(ndata*sxy-sx*sy)/del;
    for (j=1;j<=ndata;j++)
        chisq += (temp=y[j]-(aa+bb*x[j]),temp*temp);
    sigb=sqrt(chisq/del);               标准差给出迭代步长应该多大的概念
    b1=bb;
    f1=rofunc(b1);
    b2=bb+SIGN(3.0*sigb,f1);
    沿着由f1已知的向下方向,猜测跨越3-σ的划界
    f2=rofunc(b2);
    while (f1*f2 > 0.0) {               进行划界
```

```
        bb=2.0*b2-b1;
        b1=b2;
        f1=f2;
        b2=bb;
        f2=rofunc(b2);
    }
    sigb=0.01*sigb;                    不断提高精度，直至误差是一个可忽略的标准差的数
    while (fabs(b2-b1) > sigb) {
        bb=0.5*(b1+b2);                二分法
        if (bb == b1 || bb == b2) break;
        f=rofunc(bb);
        if (f*f1 >= 0.0) {
            f1=f;
            b1=bb;
        } else {
            f2=f;
            b2=bb;
        }
    }
    *a=aa;
    *b=bb;
    *abdev=abdevt/ndata;
}

#include <math.h>
#include "nrutil.h"
#define EPS 1.0e-7

extern int ndatat;                    在 medfit 中定义
extern float *xt, *yt, aa, abdevt;

float rofunc(float b)
对于给定的 b 值计算等式(15.7.16)右端的值。和程序 medfit 的数据交换是通过全局变量。
{
    float select(unsigned long k, unsigned long n, float arr[]);
    int j;
    float *arr,d,sum=0.0;

    arr=vector(1,ndatat);
    for (j=1;j<=ndatat;j++) arr[j]=yt[j]-b*xt[j];
    if (ndatat & 1) {
        aa=select((ndatat+1)>>1,ndatat,arr);
    }
    else {
        j=ndatat >> 1;
        aa=0.5*(select(j,ndatat,arr)+select(j+1,ndatat,arr));
    }
    abdevt=0.0;
    for (j=1;j<=ndatat;j++) {
        d=yt[j]-(b*xt[j]+aa);
        abdevt += fabs(d);
        if (yt[j] != 0.0) d /= fabs(yt[j]);
        if (fabs(d) > EPS) sum += d > 0.0 ? xt[j] : -xt[j];
    }
    free_vector(arr,1,ndatat);
    return sum;
}
```

### 15.7.4 其它的稳健估计方法

有时候，要从一组数据点中估计出一些参数，而对这些参数的可能值和可能不确定性有些先验的知识。在这种情况下，可以进行一个把这些预知的信息正确考虑进去的拟合，既不是用一个预先确定的值冻结一个参数，也不是使参数完全由数据组确定。这样做的形式体系

称为"先验的协方差的运用"。

在信号处理和控制理论中有一个相关的问题，我们要在噪声中找出随时间变化的信号。如果信号由一些变化非常慢的参数表征其特性，那么**卡尔曼滤波**形式体系告诉了如何对接收到的，粗糙测量的随时间而变的信号进行处理，产生最佳的参数估计。例如，如果信号是一个频域调制的正弦信号，则缓慢变化的参数可能是瞬态频率。这种情况下的卡尔曼滤波称为**锁相环**，它装置在高级的无线电接收器回路中[3,4]。

**参考文献和进一步读物：**

Huber, P. J. 1981, *Robust Statistics* (New York: Wiley). [1]

Lanuner, R. L., and Wilkinson, G. N. (eds.) 1979, *Robustness in Statistics* (New York: Academic Press). [2]

Bryson, A. E., and Ho, Y. C. 1969, *Applied Optimal Control* (Waltham, MA: Ginn). [3]

Jazwinski, A. H. 1970, *Stochastic Processes and Filtering Theory* (New York: Academic Press). [4]

# 第十六章　常微分方程组的积分

## 16.0　引言

与常微分方程组(ODEs)有关的问题通常都可以化成一系列一阶微分方程的问题来研究。例如，一个二阶常微分方程

$$\frac{d^2y}{dx^2}+q(x)\frac{dy}{dx}=r(x) \tag{16.0.1}$$

能够化成两个一阶方程

$$\begin{aligned}\frac{dy}{dx}&=z(x)\\ \frac{dz}{dx}&=r(x)-q(x)z(x)\end{aligned} \tag{16.0.2}$$

其中 $z$ 是一个新变量。这是对任意高阶常微分方程的典型处理方法。新变量通常选择使它们互为(和方程原变量)导数。有时，在定义新变量时，也加上一些和方程有关的系数或某些自变量乘方，以减缓导致计算溢出或舍入误差的积累。可以遵循这样一条常识性的原则：如果发现在结果中，原来变量变化平稳，而辅助变量变化不规则，则应该找出原因，并重新选择合适的变量。

这样，对于一般的常微分方程便可以通过一组 $N$ 个函数 $y_i(i=1,2,\ldots,N)$的 $N$ 个联立一阶微分方程组来处理。这些微分方程可以写成下面的一般形式：

$$\frac{dy_i(x)}{dx}=f_i(x,y_1,\ldots,y_N),\quad i=1,\ldots,N \tag{16.0.3}$$

其中等式右边的函数 $f_i$ 是已知的。

常微分方程组(ODEs)的解不完全由其方程决定。要确定方程的解还需要边界值条件。边界值条件是指式(16.0.3)中的函数 $y_i$ 的值的代数条件。一般来说，它们能在某些确定的离散点得到满足，但是在这些离散点之间就不成立了，即不能由微分方程自动得到满足。边界值条件有时很简单，例如，要求定义某些变量具有某些数值；有时候却较复杂，成为一组变量之间的非线性代数方程。

通常，正是边界值条件的属性决定了哪种数值解法是可行的。边界值条件可分为下面的两大类：

- **在初始值问题**中，所有的 $y_i$ 都给出了起始值 $x_s$，希望找出在终点 $x_f$ 的 $y_i$ 值，或在某离散点列(如表列间隔上)的 $y_i$ 值。
- **在两点边界值问题**中，在多个 $x$ 处定义了边界条件。在典型情况下，有的条件在 $x_s$ 处进行定义，并给出 $x_f$ 处的隐含条件。

本章将讨论初始值问题，有关两点边界值的问题将在第十七章进行讨论，后者通常更困难一些。

解决初始值问题的基本思路是：把式(16.0.3)中的 $dy$ 和 $dx$ 用有限步长 $\Delta y$ 和 $\Delta x$ 来代替。然后等式两边同乘以 $\Delta x$。这便给出了函数随自由变量 $x$ 的"步长"$\Delta x$ 变化而变化的代数公式。在把步长取得尽可能小的极限过程中，便可以得到微分方程的一个级好的近似。理论上，实现这个过程可以用**欧拉方法**(下面的式(16.1.1))，但实际上是不实用的。欧拉方法在理论上是很重要的，许多实际的方法都归结于欧拉方法同样的思想，将导数(等式右边)乘以步长，函数相应地增加一个小增量。

这一章中，我们主要讨论对常微分程组(ODEs)求解初始值问题的三种主要的实用数值解法：

- 龙格-库塔(Runge-Kutta)法。
- 理查德森(Richaxdson)外推法，其特例为布里斯奇-斯托(Bulirsch-Stroer)法。
- 预测-校正法。

下面对这三种方法分别作一简要介绍。

1. **龙格-库塔法**通过类似欧拉方法中的步长(每次都要计算出右边的 $f$ 值)，把各个步长所提供的信息结合起来，算出有一定间隔的解，然后用所得到的信息和泰勒(Taylov)展开式匹配，以得到高阶近似解。

2. **理查德森外推法**利用了一种很有效的思想，它是在如果步长比实际值小得多时，对能得到的计算值进行外推。特殊情况下，趋近零的步长值进行外推是理想的目标，**布里斯奇-斯托法**是把取每一单个步长的特殊方法(**修正中点法**)和特别的外推法(有理函数外推法)结合起来的结果。

3. **预测-校正**法把每步计算出的结果保留下来，并用那些结果外推出下一步的结果，然后，用新点上的导数信息进行校正。这方法对于平滑函数效果最好。

龙格-库塔法用于：(i)不知道别的更好的方法；(ii)用布里斯奇-斯托法无法解决的问题；(iii)一个比较平常的，不需考虑计算效率的问题。龙格-库塔法通常总是能成功的，但是它并不是最快的方法。在对 $f_i$ 进行估值比较容易，而且只需达到一定精度($\leqslant 10^{-5}$)的时候，用这种方法通常是最快的。预测-校正法因为需要利用已计算的值，所以开始计算时比较困难，但对很多平滑函数来说，它比龙格-库塔法计算效率高一些。近几年来，布里斯奇-斯托法在很多应用中取代了预测-校正法，但布里斯奇-斯托法一揽天下的说法尚为时过早。然而实践表明，只有相当复杂的预测-校正算法才能和这种方法的效果相当，因此在本书中，我们没有给出预测-校正算法的实施。我们将在第 16.7 节中进一步讨论预测-校正算法，以便当读者遇到适当的问题时，能使用一个表述比较粗略的程序。按我们的经验，我们给出的相对较简单的龙格-库塔法和布里斯奇-斯托法程序，对大多数问题来说已经足够了。

这三种方法都可以系统化编制来监视内部数据的协调性。这样便可通过改变基本步长，自动减缓一些可避免的引入解中的数字误差(自适应地)。我们一直建议在程序中实现自适应步长控制，下面我们也将这么做。

总而言之，所有三种方法都可以用于任何初始值问题。每种方法各有其优点和缺陷，这是在使用之前必须了解的。

在这一章中，我们把程序组织成相互调用的三个嵌套级别。最低一级我们称为算法程序。它实现了方法基本公式，从 $x$ 处的因变量 $y_i$ 开始，计算因变量在 $x+h$ 处的新值。算法程序中，也提供了有关该步计算结果的近似效果信息。但程序不能自适应判定这个解可不可

以接受。

精度控制判定的代码编在步进程序中，步进程序调用算法程序，它可以拒绝接受计算的结果。设置一个较小的步长，重新调用算法程序，直至达到事先定义好的精确度为止。步进的基本任务是根据特定性能选用最大步长，只有这一点达到了，算法的威力才真正显示出来。

在步进程序之上是驱动程序，用以开始和结束积分，存储中间结果，以及作为和用户的接口。我们的驱动程序并不是标准的。读者应该把它们当作一些例子，并且能为特定的应用而修改这些程序。

在下面的程序中，**rk4**、**rkck**、**mmid**、**stoerm** 和 **simpr** 是算法程序，**rkqs**、**bsstep**、**stiff** 和 **stifbs** 是步进程序；**rkdumb** 和 **odeint** 是驱动程序。

这一章的第 16.6 节将讨论**刚性方程**(*stiff equation*)的问题，这和常微分方程以及偏微分方程有关(第十九章)。

参考文献和进一步读物：


Lambert, J. 1973, *Computational Methods in Ordinary Differential Equations*(New York: Wiley).

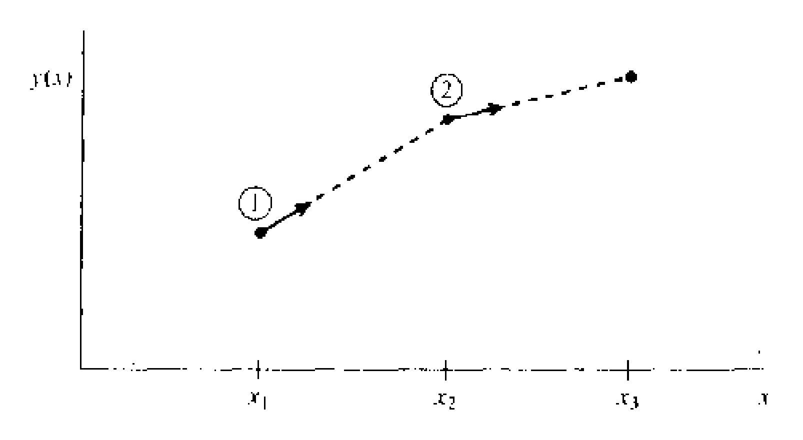
Lapidus, L., and Seinfeld, J. 1971, *Numerical Solution of Ordinary Differential Equations*(New York: Academic Press).


## 16.1 龙格-库塔法

欧拉法的公式是

$$y_{n+1} = y_n + hf(x_n, y_n) \qquad (16.1.1)$$

这里，从 $x_n$ 导出 $x_{n+1}$ 为：$x_{n+1}=x_n+h$。这个公式不是对称的：通过每次增加间隔 $h$ 将解推进，但只是在间隔的起始处用到求导的信息(参见图16.1.1)。这就意味着，引入式(16.1.1)的步长误差比修正值小 h 的一次幂，即 $O(h^2)$(可以通过幂级数展开式来证明)。




在这种求解常微分方程最简单(也是最不精确)的方法中，通过对每段间隔起始点外推来求出下一个函数值，这种方法的精确度是一阶的。

图16.1.1 欧拉方法

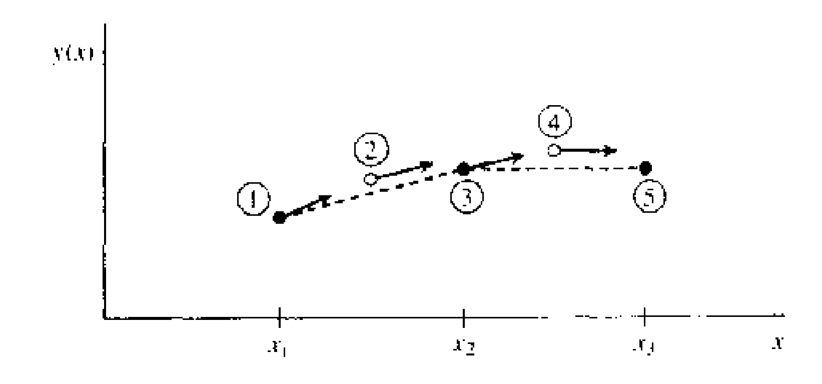
欧拉法未得到实用有几个原因，其中：(i)这种方法和其它更巧妙的方法相比，它不是非

常精确的；(ii)这种方法的稳定性也不是很好的(参见下面的第16.6节)。

但是，我们可以考虑用类似式(16.1.1)中的步骤，取间隔的中点为"试验"步。然后，用那中点的 $x$ 和 $y$ 值计算整个间隔的"实际"步长。图16.1.2说明了这个思想。写成方程是：

$$
\begin{aligned}
k_1 &= hf(x_n, y_n) \\
k_2 &= hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1) \qquad (16.1.2) \\
y_{n+1} &= y_n + k_2 + O(h^3)
\end{aligned}
$$

在式(16.1.2)的误差项中已表明，这种对称性算法消除了一阶误差项，使其误差成为**二阶**的(如果某种算法的误差量 $O(h^{n+1})$，则习惯上称它为 $n$ 阶的)。实际上，式(16.1.2)称为**二阶龙格-库塔法**或**中点法**。




使用在每一步上的初始导数，求出间隔一半处的点，然后用这个中点导数求出整段间隔的值，这样可以得到二阶的精确度，在这个图中，实心点代表最后的函数值，而空心点代表，一旦它们的导数被计算并使用以后，便被舍弃的函数值。

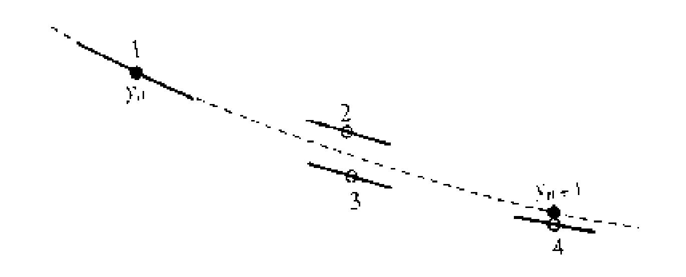
图16.1.2　中点法

下面继续讨论。有很多方法计算右边的 $f(x,y)$，它们都满足一阶误差的精度，但这些方法都各自有不同的高阶误差项。将这些方法适当结合，可以逐阶逐阶地消除误差项。这也正是龙格—库塔法的基本思想。Abramowize 和 Stegum[1]，以及 Gear[2]，都根据这种思想提出了各种不同的算式。但目前最常用、最有效的是**四阶龙格-库塔式**。这个式子有一定的光滑度：

$$
\begin{aligned}
k_1 &= hf(x_n, y_n) \\
k_2 &= hf(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}) \\
k_3 &= hf(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}) \qquad (16.1.3) \\
k_4 &= hf(x_n + h, y_n + k_3) \\
y_{n+1} &= y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5)
\end{aligned}
$$

四阶龙格-库塔法在每个步长 $h$ 中，对右端需要四次求值计算(参见图16.1.3)。如果在式(16.1.3)中，用至少可能两倍大的步长就能达到中点法式(16.1.2)一样精度的话，则这种

方法是优于中点法的。是这样的吗？答案是：通常情况是这样的，但不能说总是这样的。这让我们回到一个很重要的问题，即**高阶**并不意味着**高精度**。“四阶龙格-库塔法通常优于二阶”的说法是对的，但应该把它当作科学实践中的暂时性法则，而非严格数学中的法则。也就是说，它反映了当前科家们解决问题的一种特性。




在每一步中求导通过四步来近似：一次在起始点，两次在试探中点，一次在最末试探点。由这些导数算出最后的函数值(用实心点表示)(详细说明参见正文)。

图16.1.3 四阶龙格-库塔法

对于很多搞科研的应用者来说，四阶龙格-库塔法并不仅是常微分方程最初使用的方法，而且是现在使用的一种选择的方法。实际上，用这种古老的算法，尤其象我们下一节那样把自适应步长算法结合进去，还是能收获很大的。但是注意，这种方法也有其局限性，对于需要高精度的情况，布里斯奇-斯托法和预测-校正法会更加有效。这些方法才是真正的“赛马”，而龙格-库塔法只是用来“犁地”的马。但，老马装上新鞍也会更加灵巧。在第16.2节中，我们要给出一个龙格-库塔算法的现代实用算法，在精度要求不高时，它是很有竞争力的。在参考资料[3]中有编制龙格-库塔法程序的精彩论述。

下面是对一组几个微分方程实施一步龙格-库塔法计算的程序。输入自变量的值，然后得到一个步长 $h$(可正可负)的新值。读者会注意到，程序不但需要用户提供计算方程右端项的函数 **derivs**，还需提供起始点的导数值。为什么不让程序调用 **derivs** 来计算初始值呢？答案在下一节阐明，简言之是这样的：这个调用不只是具有这些初始条件的唯一的调用。也许先前步骤中取的步长值太大，这样便可更改。在这种情况下，就没有必要在起始点调用函数 **derivs**)。注意下面这个程序，只调用了三次 **derivs** 函数。

```
#include "nrutil.h"

void rk4(float y[], float dydx[], int n, float x, float h, float yout[],
    void (*derivs)(float, float [], float []))
    给定已知 x 的 n 个变量 y[1..n]及其导数 dydx[1..n]的值，利用四阶龙格-库塔法以间隔 h 递增求解，返回的增量
    值存储到 yout[1..n]中，它并不需和 y 矩阵有明显不同。用户提供程序 derivs(x,y,dydx)，计算在 x 处的 dydx 值。
{
    int i;
    float xh,hh,h6,*dym,*dyt,*yt;

    dym=vector(1,n);
    dyt=vector(1,n);
    yt=vector(1,n);
    hh=h*0.5;
```

```
    h6=h/6.0;
    xh=x+hh;
    for (i=1;i<=n;i++) yt[i]=y[i]+hh*dydx[i];          第一步
    (*derivs)(xh,yt,dyt);                              第二步
    for (i=1;i<=n;i++) yt[i]=y[i]+hh*dyt[i];
    (*derivs)(xh,yt,dym);                              第三步
    for (i=1;i<=n;i++) {
        yt[i]=y[i]+h*dym[i];
        dym[i] += dyt[i];
    }
    (*derivs)(x+h,yt,dyt);                             第四步
    for (i=1;i<=n;i++)                                 以适当权重积累增量
        yout[i]=y[i]+h6*(dydx[i]+dyt[i]+2.0*dym[i]);
    free_vector(yt,1,n);
    free_vector(dyt,1,n);
    free_vector(dym,1,n);
}
```

龙格-库塔法每步都采用相同的方法。以前的计算状况不用在以后的计算中，这在数学上是合适的，因为常微分方程轨迹线中的任一点都可作为初始点。这种每步处理方法都相同的这一事实，使得“驱动程序”很容易对龙格-库塔法进行调用。

对于精确计算中基本的自适应步长控制问题，我们将在下一节讨论。当然，偶尔有时只想在等距间隔上计算函数值列表，而且也不需要很高精确度。在大多数通常情况下，想要画出函数图形。这时就需要一个简单的驱动程序，以一定的步数从初始的 $x_1$ 计算到最终的 $x_f$。为了检查结果的精确性，可以加倍步数，重复计算并比较结果。这种逼近当然不会减少计算时间，而且对要求可变步长的情况也许失败。但是，这种方法减少了用户的额外考虑因素，对于小型的问题来说，这可能是考虑的最重要的一个方面。

下面是一个驱动程序，把积分的函数放在全体数组 * x 和 * * y 中；注意要分别调用 **vector()**和 **matrix()**函数为它们分配内存。

```
#include "nrutil.h"

float **y, *xx;                                      与主程序通信
void rkdumb(float vstart[], int nvar, float x1, float x2, int nstep,
    void (*derivs)(float, float [], float []))
    从已知 x1 处 nvar 个函数的初始值 vstart[1..nvar]开始，利用四阶龙格-库塔法，以相等增量递增 nstep 步到 x2，用
    户提供的 derivs(x,y,dydx)估算导数值，每一步的结果存储在全程变量 xx[1..nstep+1]，y[1..nvar][1..nstep+
    1]中。
{
    void rk4(float y[], float dydx[], int n, float x, float h, float yout[],
        void (*derivs)(float, float [], float []));
    int i,k;
    float x,h;
    float *v,*vout,*dv;

    v=vector(1,nvar);
    vout=vector(1,nvar);
    dv=vector(1,nvar);
    for (i=1;i<=nvar;i++) {                  装载起始值
        v[i]=vstart[i];
        y[i][1]=v[i];
    }
    xx[1]=x1;
    x=x1;
    h=(x2-x1)/nstep;
    for (k=1;k<=nstep;k++) {                 进行 nstep 步计算
        (*derivs)(x,v,dv);
        rk4(v,dv,nvar,x,h,vout,derivs);
```

```
        if ((float)(x+h) == x) nrerror("Step size too small in routine rkdumb");
        x += h;
        xx[k+1]=x;                    存贮中间步长
        for (i=1;i<=nvar;i++) {
            v[i]=vout[i];
            y[i][k+1]=v[i];
        }
    }
    free_vector(dv,1,nvar);
    free_vector(vout,1,nvar);
    free_vector(v,1,nvar);
}
```

参考文献和进一步读物：


Abramowitz，M.，end Stegun，I. A. 1964，*Handbook of Mathematical Founctions*，Applied Mathematics Series，vol. 55（Wasshington：National Bureau of Standards；reprinted 1968 by Dover Publications，New York），§25.5. [1]

Gear，C. W. 1971，*Numerical Initial Value Problems in Ordinary Differential Equations*（Englewood Cliffs，NJ：Prentice-Hall），Chapter 2. [2]

Shampine，L. F.，and Watts，H. A. 1977，in *Mathematical Software III*，*J. R. Rice*，*ed.*（New York：Academic Press），pp. 257-275；1979，*Applied Mathematics and Compuation*，vol. 5，pp. 93-121. [3]


## 16.2 龙格-库塔法的自适应步长控制

一个好的常微分方程的积分解法，应在求解过程中加上一些自适应控制，不断地修改步长值。通常情况下，自适应步长的控制是为了用尽可能少的计算量达到解的预定精确度。在一些不可靠的地方，应该用一些较小的步长，而在一些平滑的无关紧要的地方，则用大一些的步长以加快计算过程。其结果的效率不只是百分之几或几倍，有时是十几倍，上百倍，甚至更多。所需的精确度不仅是指在解中，有时是指在某些可被监测的相关量中。

自适应步长控制的实现，需要关于每步执行算法过程中的有关信息，尤其是截断误差有效的估计。本节中，我们将要了解能得到多少信息。显然，这些信息的计算会增加额外的计算开销，但是这种投资通常是能很快偿还的。

对于四阶龙格-库塔法，最直接的方法是步长数加倍（参见参考资料[1]）。我们把每一步分成两次计算，第一次作为整个一步，然后再一次按独立地分成两个半步来计算（参见图16.2.1）。在这种情况下，右边式求值需要进行多少额外的计算呢？每三个龙格-库塔步在求解计算中需要四次的估算，而每一步经过步长数加倍以后共用一个的起始点，所以总数是 11。因为我们用步长控制法可达到较少步长（原步长的一半）时的精度，所以应和 8 比而不是和 4 比，额外计算系数是 1.375 。这对我们能有多大损失呢？

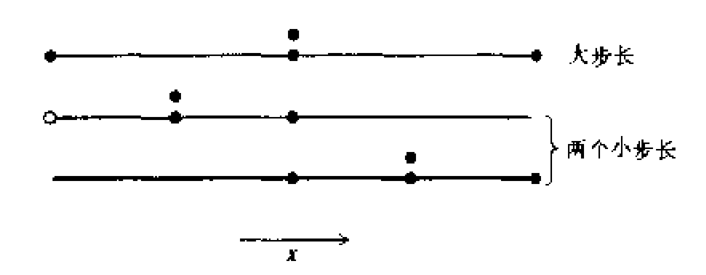
我们现在精确地论证一下从 $x$ 到 $x+2h$ 计算 $y(x+2h)$，用两种近似解法：$y_1$（每一步长 $2h$）和 $y_2$（两个步长，每个步长 $h$），因为基本方法是四阶的，两种数值近似的结果是

$$
\begin{aligned}
y(x+2h) &= y_1 + (2h)^5\varphi + O(h^6) + \dots \\
y(x+2h) &= y_2 + 2(h^5)\varphi + O(h^6) + \dots
\end{aligned}
\qquad (16.2.1)
$$

这里，对于 $h^5$，$\varphi$的值相对于步长保持为常数（根据泰劳展开式，$\varphi$的值为 $y^{(5)}(x)/5!$ 量级）。在式(16.2.1)的第一个表达式中，因为步长是$2h$，所以是$(2h)^5\varphi$，而第二个表达式中，每一步步长的误差是 $h^5\varphi$，所以结果是$2h^5\varphi$，两种数值估算的差很方便地表明了截断误差

$$\Delta \equiv y_2 - y_1 \tag{16.2.2}$$

我们对这个差值应保持恰当的精确度，不要太大也不要太小。我们可以通过调整 $h$ 来做到这一点。




实心点代表进行求导估算的点，空心点表示上一步进行过求导估算以本来是实心的点，所以估算计算的总数是每两步11次，通过比较大步长和两个小步长的精确度，来决定是在下一步调整步长值，还是因为精度不够重新计算这一点的结果。

图16.2.1　四阶龙格—库塔法通过步长数加倍进行自适应步长控制

读者也许会想到，忽略了 $h^5$以及更高阶项，我们可以通过解(16.2.1)式中的两个方程来改善理想值 $y(x+2h)$的数值估计情况，如

$$y(x+2h) = y_2 + \frac{\Delta}{15} + O(h^6) \tag{16.2.3}$$

这种近似精确到五阶，比最初的龙格-库塔法高一阶。但是我们不能想当然地这么做。式(16.2.3)也许是五阶精度，但是我们没有办法监测其截尾误差。更高阶并不代表永远是高的精确度！使用式(16.2.3)一般不会有多大坏处，但是我们无法直接了解到改善情况如何。因此我们应该用 $\Delta$ 作为误差估计，而使用式(16.2.3)计算以得到较高的精确度。使用类似式(16.2.3)的过程，技术上称为“局部外插”。

可选的步长调节算法是基于嵌入的龙格-库塔公式，最初是菲尔贝格(Fehlberg)发明的。一个很有意思的情况是，在龙格-库塔公式中，对于高于 4 的阶次 $M$ 来说，需要大于 $M$ 个函数的估计值(尽管不会大于 $M+2$)。这说明了四阶算法的通用性：给不拘一格的想法反戈一击。但是，菲尔贝格发现了用六个函数估值的五阶算法，同时六个函数估值的另外一种组合将导出了一个四阶算法，两个对 $y(x+h)$的估计值的差异能够用来估计截断误差并以调整步长。从菲尔贝格的最初的公式发现以来，另外也发现了几个其它的嵌入龙格-库塔公式。

很多实践者一度对龙格-库塔-菲尔贝格算法感到担心。这是因为用同样的估值点向前估测函数值和估计误差，比加倍步长的方法要危险得多。但是，经验表明，这种考虑实际上是不必要的。一般说来，嵌入的龙格-库塔公式，比基于加倍步长算法要提高效率一到两倍左右。

一般的五阶龙格-库塔公式是：

$$k_1 = hf(x_n, y_n)$$

$$k_2 = hf(x_n + a_2 h, y_n + b_{21} k_1)$$

$$\cdots$$

$$k_6 = hf(x_n + a_6h, y_n + b_{61}k_1 + \cdots + b_{65}k_5)$$
$$y_{n+1} = y_n + c_1k_1 + c_2k_2 + c_3k_3 + c_4k_4 + c_5k_5 + c_6k_6 + O(h^6) \tag{16.2.4}$$

嵌入四阶公式是：

$$y_{n+1}^* = y_n + c_1^*k_1 + c_2^*k_2 + c_3^*k_3 + c_4^*k_4 + c_5^*k_5 + c_6^*k_6 + O(h^5) \tag{16.2.5}$$

误差估计值为：

$$\Delta = y_{n+1} - y_{n+1}^* = \sum_{i=1}^{6}(c_i - c_i^*)k_i \tag{16.2.6}$$

我们倾向于使用不同常量的特殊值，凯希(**Cash**)和卡珀(**Karp**)[2]的发现，列在表16.2.1中。这给出了比菲尔贝格原来值更有效的算法，在误差特性方面更为好一些。

**表16.2.1　嵌入的龙格-库塔的凯希-卡珀参数**

| $i$ | $a_i$ | $b_{ij}$ | | | | | $c_i$ | $c_i^*$ |
|---|---|---|---|---|---|---|---|---|
| 1 | | | | | | | $\frac{37}{378}$ | $\frac{2825}{27648}$ |
| 2 | $\frac{1}{5}$ | $\frac{1}{5}$ | | | | | 0 | 0 |
| 3 | $\frac{3}{10}$ | $\frac{3}{40}$ | $\frac{9}{40}$ | | | | $\frac{250}{621}$ | $\frac{18575}{48384}$ |
| 4 | $\frac{3}{5}$ | $\frac{3}{10}$ | $-\frac{9}{10}$ | $\frac{6}{5}$ | | | $\frac{125}{594}$ | $\frac{13525}{55296}$ |
| 5 | 1 | $-\frac{11}{54}$ | $\frac{5}{2}$ | $-\frac{70}{27}$ | $\frac{35}{27}$ | | 0 | $\frac{277}{14336}$ |
| 6 | $\frac{7}{8}$ | $\frac{1631}{55296}$ | $\frac{175}{512}$ | $\frac{575}{13824}$ | $\frac{44275}{110592}$ | $\frac{253}{4096}$ | $\frac{512}{1771}$ | $\frac{1}{4}$ |
| $j=$ | | 1 | 2 | 3 | 4 | 5 | | |

现在，我们至少已大约知道误差的所在，我们需要考虑如何将其控制在一定范围以内。$\Delta$ 和 $h$ 之间有什么联系吗？根据式(16.2.1)～(16.2.2)可知，$\Delta$ 和 $h^5$ 成正比。如果我们取步长 $h_1$，产生误差 $\Delta_1$，则取步长 $h_0$ 时，产生的误差可以估计为

$$h_0 = h_1\left|\frac{\Delta_0}{\Delta_1}\right|^{0.2} \tag{16.2.7}$$

因此，我们用 $\Delta_0$ 来表示所需的精确度。这样等式(16.2.7)可用于两个方面：如果 $\Delta_1$ 数值上比 $\Delta_0$ 大，则从等式可以得出，再次试当前(失败)步时应该把步长减少多少；如果 $\Delta_1$ 和 $\Delta_0$ 小，则从等式可以得出，在下一步时我们提高步长值多大是安全的。局部外插接受五阶的值 $y_{n+1}$，尽管误差估计实际上用于四阶值 $y_{n+1}^*$。

我们的表示式中隐藏着这样一个事实：$\Delta_0$ 实际上是所需精确度的一个向量，每一组常微分方程的每个等式都有一个误差项。总的说来，我们的精度要求，各等式的误差都在各自允许的误差范围以内。也就是说，我们可根据“最坏情况”等式的需求相应地放缩步长大小。

我们怎么把所需要的精度，例如，叙述为“精确到$10^6$分之一”，和 $\Delta_0$ 联系来呢？这是一个比较难以确定的问题，这和实际应用问题有关！有时也许在处理一组因变量数值相差很大的方程，这种情况下，也许希望用相对误差，$\Delta_0 = \epsilon y$，这里 $\epsilon$ 是类似$10^{-6}$或其它别的什么样的

值。有时也许在处理某些振荡函数，穿过零点但限制于某一最大值范围内，这时，也许把 $\Delta$ 设置成最大值的 $\epsilon$ 倍。

综合这些考虑设计出一个比较通用的程序。程序的一个参量，当然是在当前起始步中因变量的向量，把这称为 y[1..n]。再让用户为每一步，定义另外一个相应的向量参量 yscal[1..n]，以及总的容差限 eps。这样，对于第 $i$ 方程来说所需的精确度就为

$$\Delta_0 = \text{eps} \times \text{yscal}[i] \tag{16.2.8}$$

如果想用相对误差表示，则可以把指向 y 的指针放入 yscal 指针的调用位置(不需要把值拷贝到另外一个矩阵中)。如果读者想用与某一最大值有关的绝对误差形式，则可以把 yscal 的元素设置成那些最大值。对于那些不是非常靠近过 0 点的情况，有一种很有用的"诀窍"可以得到恒定的相对误差，那就是把 yscal[i]设置成和 |y[i]|+|$h$×dydx[i]|相等即可(下面的程序 **odeint** 便是这样处理的)。

下面是更关键的技术要点。我们必须考虑 yscal 的另外一种可能性。目前我们所提到误差准则都是"局部"的，是考虑每一步的误差。在有的实际情况下，通常想计算"全程"累积误差，从积分的起点到终点；以及在最坏情况下，所有的误差都取同一符号而进行累加。这样，步长 h 越小，所要处理的值 $\Delta_0$ 也越小。为什么?因为从起点到终点，$x$ 会有更多的步数。在这种情况下，就需要让 yscal 和 $h$ 成比例，得到类似下面的式子：

$$\Delta_0 = \epsilon h \times \text{dydx}[i] \tag{16.2.9}$$

这使使相对精度 $\epsilon$ 不是作用于 y，而是(更加严格一些)作用于每一步值的增量。我们回头看一看式(16.2.7)。如果 $\Delta_0$ 隐含和 $h$ 成比例，则指数 0.2 就不再正确了：当步长值从一个很大的值减小时，当 yscal 也随新的预测值 $h_1$ 改变时，这个 $h_1$ 便达不到所需的精度。因此我们必须把原来的 0.2=1/5 放大到 0.25=1/4 才能正常使用。

指数 0.20 和 0.25 实际上并没有很大不同。这促使我们采纳了下面更合理的逼近方法，作为用户也就没有必要事先知道用步长来放缩 **yscal** 的值。无论何时减小步长，我们都用较大的指数值；(不管需要不需要!)而增加步长时，用较小一些的指数值。而且，因为我们对误差的估计不是精确的，只和 $h$ 的主项阶的精度有关，所以建议加入一个安全系数 $S$ 。$S$ 的值比单位 1 小百分之几。这样，方程(16.2.7)便替换成：

$$h_0 = \begin{cases} Sh_1 \left| \dfrac{\Delta_0}{\Delta_1} \right|^{0.20} & \Delta_0 \geq \Delta_1 \\ Sh_1 \left| \dfrac{\Delta_0}{\Delta_1} \right|^{0.25} & \Delta_0 < \Delta_1 \end{cases} \tag{16.2.10}$$

在实际应用中我们发现这种表示是有效的。

下面是采用了"控制计算质量"的龙格-库塔法的步进程序。

```
#include <math.h>
#include "nrutil.h"
#define SAFETY 0.9
#define PGROW -0.2
#define PSHRNK -0.25
#define ERRCON 1.89e-4          ERRCON 的值等于(5/SAFETY)的(1/PGROW)次幂

void rkqs(float y[], float dydx[], int n, float *x, float htry, float eps, float yscal[], float *hdid,
    float *hnext, void (*derivs)(float, float[], float []))
    五阶龙格-库塔法具有截断误差的监控，以保证精度和调整步长。输入的是在独立变量 x 处的初始值对应的因变向
```

量 y[1..n]和其导数 dydx[1..n]，尝试的步长为 htry，所需精度为 eps，对误差进行标度的向量为 yscal[1..n]。输出中，y 和 x 由其新值替换，hdid 是实际采用的步长，hnext 是估算的下一步的步长值，desive 是由用户提供的程序，用于计算右边的导数。

```
{
    void rkck(float y[], float dydx[], int n, float x, float h,
        float yout[], float yerr[], void (*derivs)(float, float [], float []));
    int i;
    float errmax,h,htemp,xnew,*yerr,*ytemp;

    yerr=vector(1,n);
    ytemp=vector(1,n);
    h=htry;                              设置步长为初始实验值
    for (;;) {
        rkck(y,dydx,n,*x,h,ytemp,yerr,derivs);        进行一步
        errmax=0.0;                      估计精度
        for (i=1;i<=n;i++) errmax=FMAX(errmax,fabs(yerr[i]/yscal[i]));
        errmax /= eps;                   选择相应的容限
        if (errmax > 1.0) {              截断误差太大，减小步长
            htemp=SAFETY*h*pow(errmax,PSHRNK);
            h=(h >= 0.0 ? FMAX(htemp,0.1*h) : FMIN(htemp,0.1*h));
            小于10的系数
            xnew=(*x)+h;
            if (xnew == *x) nrerror("stepsize underflow in rkqs");
            continue;                    再试验一次
        } else {                         成功，计算下一步大小
            if (errmax > ERRCON) *hnext=SAFETY*h*pow(errmax,PGROW);
            else *hnext=5.0*h;           不大于5增量
            *x += (*hdid=h);
            for (i=1;i<=n;i++) y[i]=ytemp[i];
            break;
        }
    }
    free_vector(ytemp,1,n);
    free_vector(yerr,1,n);
}
```

程序 nkqs 要调用程序 rkck，它采用凯希-卡珀-龙格-库塔步骤：

```
#include "nrutil.h"

void rkck(float y[], float dydx[], int x, float x, float h,float yout[],
    float yerr[], void (*derivs)(float, float[], float []))
```

给定在各 x 点的处的 n 个变量 y[1..n]及其导数值 dydx[1..n]，用五阶凯希-卡珀-龙格-库塔法对间隔 h 推导解，并返回增加的变量 yout[1..n]。同时，用嵌入四阶算法将局部截断误差返回到 yout 中。用户提供程序 derivs(x,y,dydx)，它返回 x 点的导数值。

```
{
    int i;
    static float a2=0.2,a3=0.3,a4=0.6,a5=1.0,a6=0.875,b21=0.2,
        b31=3.0/40.0,b32=9.0/40.0,b41=0.3,b42 = -0.9,b43=1.2,
        b51 = -11.0/54.0, b52=2.5,b53 = -70.0/27.0,b54=35.0/27.0,
        b61=1631.0/55296.0,b62=175.0/512.0,b63=575.0/13824.0,
        b64=44275.0/110592.0,b65=253.0/4096.0,c1=37.0/378.0,
        c3=250.0/621.0,c4=125.0/594.0,c6=512.0/1771.0,
        dc5 = -277.00/14336.0;
    float dc1=c1-2825.0/27648.0,dc3=c3-18575.0/48384.0,
        dc4=c4-13525.0/55296.0,dc6=c6-0.25;
    float *ak2,*ak3,*ak4,*ak5,*ak6,*ytemp;

    ak2=vector(1,n);
    ak3=vector(1,n);
```

```
    ak4=vector(1,n);
    ak5=vector(1,n);
    ak6=vector(1,n);
    ytemp=vector(1,n);
    for (i=1;i<=n;i++)                          第一步
        ytemp[i]=y[i]+b21*h*dydx[i];
    (*derivs)(x+a2*h,ytemp,ak2);                第二步
    for (i=1;i<=n;i++)
        ytemp[i]=y[i]+h*(b31*dydx[i]+b32*ak2[i]);
    (*derivs)(x+a3*h,ytemp,ak3);                第三步
    for (i=1;i<=n;i++)
        ytemp[i]=y[i]+h*(b41*dydx[i]+b42*ak2[i]+b43*ak3[i]);
    (*derivs)(x+a4*h,ytemp,ak4);                第四步
    for (i=1;i<=n;i++)
        ytemp[i]=y[i]+h*(b51*dydx[i]+b52*ak2[i]+b53*ak3[i]+b54*ak4[i]);
    (*derivs)(x+a5*h,ytemp,ak5);                第五步
    for (i=1;i<=n;i++)
        ytemp[i]=y[i]+h*(b61*dydx[i]+b62*ak2[i]+b63*ak3[i]+b64*ak4[i]+b65*ak5[i]);
    (*derivs)(x+a6*h,ytemp,ak6);                第六步
    for (i=1;i<=n;i++)                          用适当的权重累积增量
        yout[i]=y[i]+h*(c1*dydx[i]+c3*ak3[i]+c4*ak4[i]+c6*ak6[i]);
    for (i=1;i<=n;i++)
        yerr[i]=h*(dc1*dydx[i]+dc3*ak3[i]+dc4*ak4[i]+dc5*ak5[i]+dc6*ak6[i]);
        估计作为四阶和五阶算法的误差
    free_vector(ytemp,1,n);
    free_vector(ak6,1,n);
    free_vector(ak5,1,n);
    free_vector(ak4,1,n);
    free_vector(ak3,1,n);
    free_vector(ak2,1,n);
}
```

注意，上面的程序都是单精度的，在定义 eps 时不要太贪心了。贪心会得到惩罚的。应和用基尔伯特(Gillest)和萨利文(Sullivan)法则：程序把步长取得很小，使各个 hy' 加上各个 y，就象加了零一样，程序始终能取得近似的误差。于是程序来回地运算，选用无穷多个小步长，而丝毫不会改变一点点因变量的值。(为了避免这种灾难性的后果，可以监视是否出现异常小的步长，或一步步计算时因变量值是不是没有变化。在微机上，可以判定执行这个程序的时间不会超过一顿午饭时间。)

下面是一个成熟的自适应步长控制龙格-库格算法的"驱动程序"。我们热诚地推荐用这个程序或类似的程序来处理各种问题，包括普通常微分方程和常微分方程组，以及有限积分(增强了第四章中这种方法的功能)。对于中间结果的存储(如果想查看)，我们假定全程指针参数 * xp 和 * * yp 已被有效初始化(例如通过应用程序 vector 和 matrix())，因为每一步的间隔是不等的，结果只能在比 dxsav 大的间隔中存储起来。全程变量 kmax 指明了能够存储的更大的步数。如果 kmax＝0，则没有中间存储区，而且指针 * xp 和 * * yp 不需要指向有效内存。如果超过了 kmax 值，便停止存储步长，除了终止值总要被存储的情况以外。而且这些控制只是为用户的需求而指定的。**odeint** 应该常用于解决手头的问题。

```
#include <math.h>
#include "nrutil.h"
#define MAXSTP 10000
#define TINY 1.0e-30

extern int kmax,kount;
extern float *xp, **yp, dxsav;
```

为中间结果所需的存储空间。在程序调用中预置 kmax 和 dxsav。若 kmax≠0，结果将以 dxsav 的近似间隔存入数组 xp[1..kount]，yp[1..nvar][]中，其中 kount 是程序 odeint 的输出，对这些变量的定义说明，对数组具有 xp[1..kmax]

和 yp[1..nvar][1..kmax]的内存分配，都将在程序调用时给出。

void odeint(float ystart[], int nvar, float x1, float x2, float eps, float h1,
float hmin, int *nok, int *nbad, void (*derivs)(float, float [], float []),
void (*rkqs)(float [], float [], int, float *, float, float, float[],
float *, float *, void (*)(float, float[], float [])))

具有自适应控制的龙格-库塔法驱动程序。积分的初值 ystart[1..nvar]从 x1到 x2，精度为 eps，在全程变量中存储中间结果。h1设置成第一步的预测步长，hmin 作为步长的最低限(可以是0)，对于输出，nok 和 nbad 是进行的成功或失败(但重新计算和调整)的步数，ystart 由积分间隔末尾的值来取代。derivs 是用户提供用来计算右边求导结果的程序，rkqc 是所用的步进程序。

```
{
    int nstp,i;
    float xsav,x,hnext,hdid,h;
    float *yscal,*y,*dydx,

    yscal=vector(1,nvar);
    y=vector(1,nvar);
    dydx=vector(1,nvar);
    x=x1;
    h=SIGN(h1,x2-x1);
    *nok = (*nbad) = kount = 0;
    for (i=1;i<=nvar;i++) y[i]=ystart[i];
    if (kmax > 0) xsav=x-dxsav*2.0;             保证第一步的存贮空间
    for (nstp=1;nstp<=MAXSTP;nstp++) {          最大步数为MAXSTP
        (*derivs)(x,y,dydx);
        for (i=1;i<=nvar;i++)
          用于监测精确度的放缩，如果需要的话，这个选择可以进行修改

            yscal[i]=fabs(y[i])+fabs(dydx[i]*h)+TINY;
        if (kmax > 0 && kount < kmax-1 && fabs(x-xsav) > fabs(dxsav)) {
            xp[++kount]=x;                       存入中间结果
            for (i=1;i<=nvar;i++) yp[i][kount]=y[i];
            xsav=x;
        }
        if ((x+h-x2)*(x+h-x1) > 0.0) h=x2-x;     如果步长超出尾端，则缩短步长
        (*rkqs)(y,dydx,nvar,&x,h,eps,yscal,&hdid,&hnext,derivs);
        if (hdid == h) ++(*nok); else ++(*nbad);
        if ((x-x2)*(x2-x1) >= 0.0) {             完成了吗
            for (i=1;i<=nvar;i++) ystart[i]=y[i];
            if (kmax) {
                xp[++kount]=x;                   存储最后一步
                for (i=1;i<=nvar;i++) yp[i][kount]=y[i];
            }
            free_vector(dydx,1,nvar);
            free_vector(y,1,nvar);
            free_vector(yscal,1,nvar);
            return;                              正常退出
        }
        if (fabs(hnext) <= hmin) nrerror("Step size too small in odeint");
        h=hnext;
    }
    nrerror("Too many steps in routine odeint");
}
```

## 参考文献和进一步读物：


Gear, C. W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Cash, J. R., and Karp, A. H. 1990, *ACM Transactions on Mathematical Software*, vol. 16, pp. 201～222. [2]

## 16.3 修正中点法

这一节讨论**修正中点法**。从点 $x$ 到点 $x+H$ 通过每一步长为 $h$ 的 $n$ 子步来计算因变量 $y(x)$ 向量，其中

$$h = H/n \tag{16.3.1}$$

原则上，可用修正中点算法作为一种常微分方程的积分器。实际上，这种方法最重要的应用是用在第16.4节介绍的布里斯基-斯托法中。因此，也可以把这一节当作第16.4节的一个前奏。

等式右边所需的估值计算，在修正中点法中需要 $n+1$ 次。这种算法的公式如下

$$\begin{aligned} z_0 &\equiv y(x) \\ z_1 &= z_0 + hf(x,z_0) \\ z_{m+1} &= z_{m-1} + 2hf(x+mh,z_m) \qquad m=1,2,\ldots,n-1 \\ y(x+H) \approx y_n &\equiv \frac{1}{2}[z_n - z_{n-1} - hf(x+H,z_n)] \end{aligned} \tag{16.3.2}$$

这里所有 $z$ 是沿每步 $h$ 的中间近似值，而 $y_n$ 是用于近似 $y(x+H)$ 的结果。除第一点和最后一点以外，这种算法根本说来是"中心差分"或"中点"法（和等式(16.1.2)相比），因此给了个限制词"修正"。

修正中点法是一种二阶算法，象式(16.1.2)一样，但是有一优点，每一步 $h$ 只需要（对于比较大的 $n$）一个导数值，而不象二阶龙格-库塔法中需要两个导数值。也许因为式(16.3.2)的简单性，也很容易嵌入某些应用程序中。但是总的说来，修正中点法不只用在前一节中实施的自适应控制的四阶龙格-库塔算法。

在布里斯奇-斯托法中，用修正中点法是为了从式(16.3.2)中求出一"深刻"的解来。这已由 Gzagg 证明，所得出的式(16.3.2)的误差可表达为 h 的一个级数，而且只含 $h$ 的偶次项

$$y_n - y(x+H) = \sum_{i=1}^{\infty} \alpha_i h^{2i} \tag{16.3.3}$$

其中 $H$ 保持常量，但在式(16.3.1)中 $h$ 随 $n$ 而变。这种偶次方级数的重要性在于，如果我们仍运用通过组合步长的情况来达到消去高阶误差项的话，每一次我们可以提高两阶！

例如，假设 $n$ 是偶的，并设 $y_{n/2}$ 表示应用式(16.3.1)和(16.3.2)，且步数为一半 n→n/2 的结果。则估计式

$$y(x+H) \approx \frac{4y_n - y_{n/2}}{3} \tag{16.3.4}$$

是四阶的精确度，和四阶龙格-库塔法一样，但每步 $h$ 只需要1.5次求导估值，而不是龙格-库塔的四次。暂时不要着急实现式(16.3.4)，我们还能作得更好些。

现在我们最好回过头来看一看第4.2节中的程序 **qsimp**，并把等式(4.2.4)和上面的式(16.3.4)进行一下比较，就会发现在第四章中，根据理查德森外推思想进行的变换，在第4.3节龙贝格(Ronlesg)积分中实现，恰好类似于从这一节到下一节的转移。

下面是修正中点算法的实现，在以后将要用到。

```
#include "nrutil.h"

void mmid(float y[], float dydx [], int nvar, float xs, float htot, int nstep,
    float yout[], void (*derivs)(float, float[], float[]))
```

修改中点法，在 xs，输入因变量向量 y[1..nvar]和它的求导向量 dydx[1..nvar]。输入还有总的步数 htot，以及每一步中的子步的数目 nstep，输出以 yout[1..nvar]返回，该矩阵不必和 y 一样；但如果和 y 不一样，程序不必完好无损地返回 y 和 dydx 值。devirs 是用户提供用于计算右边求导值的程序。

```
{
    int n,i;
    float x,swap,h2,h,*ym,*yn;
     ym=vector(1,nvar);
     yn=vector(1,nvar);
     h=htot/nstep;                          步长的行程
     for (i=1;i<=nvar;i++) {
         ym[i]=y[i];
         yn[i]=y[i]+h*dydx[i];              第一步
     }
     x=xs+h;
     (*derivs)(x,yn,yout);                  用yout暂存导数值
     h2=2.0*h;
     for (n=2;n<=nstep;n++) {               一般步
         for (i=1;i<=nvar;i++) {
             swap=ym[i]+h2*yout[i];
             ym[i]=yn[i];
             yn[i]=swap;
         }
         x += h;
         (*derivs)(x,yn,yout);
     }
     for (i=1;i<=nvar;i++)                  最后一步
         yout[i]=0.5*(ym[i]+yn[i]+h*yout[i]);
     free_vector(yn,1,nvar);
     free_vector(ym,1,nvar);
}
```

## 16.4 理查德森外推法和布里斯奇-斯托算法

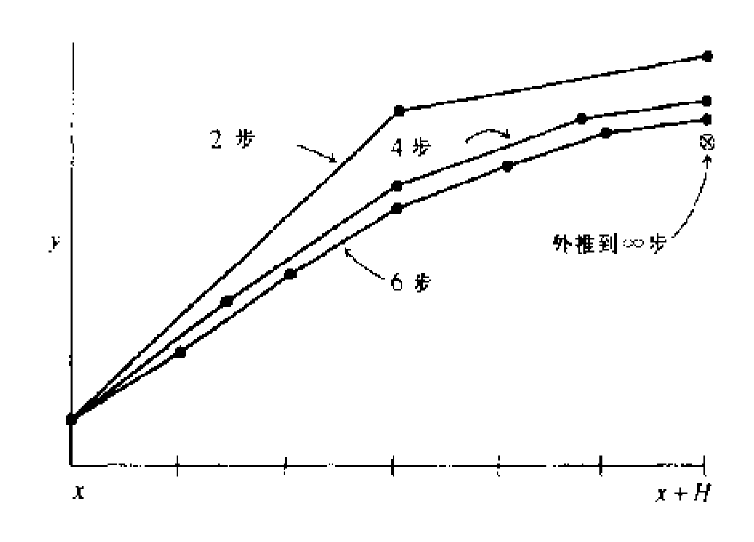
在这一节的介绍的方法不是针对包括非平滑函数的微分方程组的。例如，读者也许会碰到这样的差分方程，方程右边的函数值是通过查找表或插值来定义的。如果是这样的话，可以回到龙格-库塔法用自适应步长来计算。那种方法对于不固定的或不连续的定义域的情况处理得很好。同时也是一种对微分方程的一种快速而较粗糙，精度不是很高的解法。另外一点要说明的是，在这一节介绍的方法对于在积分区间中有奇异点的微分方程处理得不是特别好。一个正规解需要很小心地一点一点逼近这些点，而带自适应步长的龙格-库塔法有时处理这个问题很有效；更普遍的是，采用其它特别的方法来解决这些问题，但这已超出了我们这里范围。

除了上边的两个缺点以外，我们认为这一节中的布里斯奇-斯托法是用最小的计算达到常微分方程求解高精度的一个目前来说最好的方法(有时可能有例外，这种例外在实际中很少发生，将在下一部分讨论)。

这个方法有三个关键，第一个是**理查德森延迟逼近极限**，我们已在第4.3节中的龙贝格积分中已经遇到了。其思想是把数值计算的结果，作为某一象步长 $h$ 这样的可调整参数的解析函数(如果是一个比较复杂的)。这个解析函数可以通过进行不同 $h$ 值的计算来估测，这里的 $h$ 值没有必要为达到我们所需的精度而取得太小。当我们得到这个函数以后，我们把函数拟合成某个解析式，然后在很重要的点 $h=0$ 处进行估算(参见图16.4.1)。理查德森外推法

真是一种点石成金的方法！

第二个关键是用什么样的匹配函数。布里斯基和斯托首先认识到了在理查德森方法中**有理函数外推的作用**。有理函数的外推突破了幂级数和其收敛半径的限制，只受复平面的第一次极点的影响，即使在 $h$ 的各次项数量级相差不大的情况下，有理函数匹配仍能很好地近似解析函数。换言之，$h$ 可以取得较大，让这种方法的"阶次"失去意义，而这种方法仍能取得很好的效果。关于这一点可以重新看一下第3.1~3.2节，在那里我们已经讨论过有理函数外推的问题。




一个大的间隔 H 由逐渐细分的子步序列来组成，其结果是对设想的无穷细分的子步进行外推。在布里斯奇-斯托法中，积分法是采用修正中点法，外推法是采用有理函数的外推法。

图16.4.1　布里斯奇-斯托法中所用的理查德森外推法

第三个关键在前一节已经讨论过，也就是指让误差函数是严格偶函数，使有理函数或多项式逼近为变量 $h^2$ 的形式而非 $h$ 的形式。

把这三个主要思想综合在一起，我们便得出了**布里斯奇-斯托法**[1]。单步的布里斯奇-斯托法是从 $x$ 计算到 $x+H$，这里的 H 是一个比较大——而不是近似无穷小——的距离。每一单步是由很多(几百上千个)修正中点法的子步组成，它们都采用近似于零步长的有理函数的外推算法。估算间隔 $H$ 的序列是由 $n$(子步长数目)的递增序列组成的。通常布里斯奇和斯托提出的 $n$ 序列为

$$n = 2,4,6,8,12,16,24,32,48,64,96,\ldots,[n_j = 2n_{j-2}],\ldots \qquad (16.4.1)$$

Deuflhard[2,3]最近发表的文章中建议使用序列

$$n = 2,4,6,8,10,12,14,\ldots,[n_j = 2j],\ldots \qquad (16.4.2)$$

通常它更为有效。对于每一步来说，我们事先不知道沿这个序列要执行到多远。在顺序执行每一个 $n$ 值以后，我们企图用有理外推函数进行试验。外推结果给出了外推值和误差估计。如果误差不满足要求，则采用更大的 $n$ 值；如果误差满足要求，则继续下一步，从 $n=2$ 重新开始。

当然必须有一个上限，超过上限以后，我们可得出结论，在沿间隔 H 以内的路径中有障

得，因此我们必须减少 H 值而不是对 H 进行更细的划分。在下面的程序实施中，所要试的 n 的最大数目称为 KMAXX。我们通常取这个值等于 8，因为式(16.4.2)的序列中第 8 个值是 16，所以这就是我们允许 $H$ 的最大细划分数。

和龙格-库塔算法一样，这是我们同样通过监视内部变元的一致性，以及根据局部截断误差的允许误差范围，来调整步长大小以控制误差的大小。修正中点法积分的每一个新的结果允许象第3.1节中的图表那样被附加的一组对角元扩充。每一步增加的新校正量被采用为（保守的）误差估计。我们应该怎样用这个误差估值来调整步长呢？目前最好的方案是 Deuflhard[2,3]提出的。其完整内容摘录如下：

假设从外推表第 $k$ 列（也就是第 $k+1$ 行）返回的误差估计的绝对值为 $\varepsilon_{k+1,k}$，误差控制是通过在接受当前步时需要满足

$$\varepsilon_{k+1,k} < \varepsilon \tag{16.4.3}$$

来达到，其中 $\varepsilon$ 是允许的误差范围。对于式(16.4.2)的偶序列，该方法的阶次是 $2k+1$：

$$\varepsilon_{k+1,k} \sim H^{2k+1} \tag{16.4.4}$$

这样，为了在某一指定列 $k$ 达到收敛，一个新步长 $H_k$ 的简单估计值则为

$$H_k = H\left(\frac{\varepsilon}{\varepsilon_{k+1,k}}\right)^{1/(2k+1)} \tag{16.4.5}$$

我们的目的是，使哪一个 $k$ 列达到收敛性呢？我们比较一下不同 $k$ 列所需的工作量。我们假设 $A_k$ 为达到外推图表的 $k$ 行所需要的计算量，而 $A_{k+1}$ 是达到 $k$ 列的工作量，我们假定工作量是由微分方程右边的函数估值来决定的。对于 $H$ 中 $n_k$ 细分，函数估值的数目可由下面的递推式来求出

$$\begin{aligned} A_1 &= n_1 + 1 \\ A_{k+1} &= A_k + n_{k+1} \end{aligned} \tag{16.4.6}$$

达到 $k$ 列的每一小步的工作量是 $A_{k+1}/H_k$，这是我们用因子 $H$ 进行无量纲化，并写作

$$W_k = \frac{A_{k+1}}{H_k}H \tag{16.4.7}$$

$$= A_{k+1}\left(\frac{\varepsilon_{k+1,k}}{\varepsilon}\right)^{1/(2k+1)} \tag{16.4.8}$$

在积分的过程中，$W_k$ 的值可以进行计算。这样最优列 $q$ 以下标由下式定义：

$$W_q = \min_{k=1,\ldots,k_f} W_k \tag{16.4.9}$$

这里，$k_f$ 是最后一列，在这一列中应满足式(16.4.3)的误差范围。由式(16.4.9)定义的 $q$ 决定了用于下一个基本的步长值 $H_q$。这样我们可望在最优列 $q$ 处达到收敛。

对于该算法必须有两个限制：

- 如果以前的 $H$“太小”，则 $k_f$ 也会“太小”。这样，$q$ 也“太小”就可能需要增大 $H$ 值以便能在列 $q>k_f$ 上达到收敛。
- 如果当前的 H“太大”，在当前步无法收敛，则必须减小 $H$ 值。我们可以通过检测 $\varepsilon_{k+1,k}$ 的值来发现这一点，使得我们可以尽可能早地停止当前步。

Deuflhard 的文章中用通信原理中的思想，确定外推的“平均收敛期望”来处理这两个问题。在他的模型中，他提出了一个校正因子 $\alpha(k,q)$，通过和 $H_k$ 相乘以后试图在 $q$ 列处得到收敛。因子 $\alpha(k,q)$ 只依赖于 $\varepsilon$ 及序列$\{n_i\}$，这样可以在初始化的时候一次算出：

$$\alpha(k,q) = \varepsilon^{\frac{A_{k+1}-A_{q+1}}{(2k+1)(A_{q+1}-A_1+1)}} \qquad 对于 \quad k<q \tag{16.4.10}$$

而且 $\alpha(q,q)=1$。

现在处理第一个问题。假设收敛发生在列 $q=k_f$。我们希望增加步长使在 $q+1$列达到收敛，而不是用

$H_q$ 来进行下一步操作。因为我们不能计算出可用的 $H_{q-1}$ 值，故而把它估计为

$$H_{q+1} = H_q\,\alpha(q, q+1) \tag{16.4.11}$$

根据式(16.4.7)，这种替代是有效的，即可以减小每一小步的计算量。如果

$$\frac{A_{q+1}}{H_q} > \frac{A_{q+2}}{H_{q+1}} \tag{16.4.12}$$

或

$$A_{q+1}\alpha(q, q+1) > A_{q+2} \tag{16.4.13}$$

在初始化过程中，这种不等关系对于 $q=1,2,\ldots$ 可以检查出来，以决定 $k_{max}$，最大允许的列。这样，当式(16.4.12)满足时，用 $H_{q-1}$ 总是有效的(实践中我们把 $k_{max}$ 限制量为 8，即使当 $\varepsilon$ 很小时，在这种情况下有效性不会有多大改进，而舍入误差却成了问题)。

步长减小的问题可以通过在当前步中估算步长值

$$H_k = H_k\,\alpha(k, q), \qquad k = 1, \ldots, q-1 \tag{16.4.14}$$

来完成。$H$ 在最优列 $q$ 中，达到收敛的步长的估计值。如果 $H_k$"太小"，我们则不再继续当前步，而是用 $H_k$ 重新开始。"太小"由以下规则进行制定

$$H_k\alpha(k, q+1) < H \tag{16.4.15}$$

$\alpha$ 满足 $\alpha(k,q+1) > \alpha(k,q)$。

在第一步中，我们并没有关于解的任何信息，步长减小的检查是针对所有 k 来完成的。以后，我们在一个"序号窗口"中检测收敛和可能的步长减小，该"序号窗口"如下

$$\max(1, q-1) \le k \le \min(k_{max}, q+1) \tag{16.4.16}$$

之所以这样定义窗口是因为，一方面，如果对 $k<q-1$ 处似乎象收敛的样子，那往往是假的，这是在外推过程中小误差偶然作用的结果；另一方面，如果需要在 $k=q+1$ 以外的地方达到收敛，这个局部收敛模型显然不好，就需要减小步长重建此模型。

在程序 bsstep 中，这些各种各样的检测实际上是用

$$\varepsilon(k) = \frac{H}{H_k} = \left(\frac{\epsilon_{k+1,k}}{\epsilon}\right)^{1/(2k+1)} \tag{16.4.17}$$

在程序中命名为 err[k] 的值来实现的。和以往一样，我们在步长选择中包括了一个"安全系数"，这是通过用0.25ε 来代替 ε 达到的，其他的安全系数在程序注释中说明。

注意，尽管最优收敛列的限制为最多一次递增一步，但根据式(16.4.9)，顺序是允许打乱的。这就使得这种方法对不连续的问题有一定适应性。

我们再次提醒，变元的尺度比例通常对微分方程的成功求解是很关键的，在式(16.2.8)以后讨论的尺度比例技巧是一种很好的通用的选择，但并非完全正确的。以变量最大值为尺度比例是较稳健的，但需要有一些先验的信息。

下面的程序实现了布里斯奇-斯托法的一步计算，它和控制计算质量的龙格-库塔法步进程序 **rkqv** 所调用的变量序列是完全一样的。这就是说，在第16.2节中的驱动程序 **odeint** 能够象调用龙格-库塔法程序一样调用布里斯奇-斯托法程序，只需在 **odeint** 程序中将 **rkqs** 函数用 **bsstep** 替换，并注意要让程序是浮点型或双精度型保持一致，程序 **bsstep** 调用 **mmid** 采用修正中点序列，然后调用下面的 **pzextr** 进行多项式外推。

```
#include <math.h>
#include "nrutil.h"
#define KMAXX 8                          用于外推的最大行数
#define IMAXX (KMAXX+1)
```

```
#define SAFE1 0.25                       安全因子
#define SAFE2 0.7
#define REDMAX 1.0e-5                    步长减小的最大因子
#define REDMIN 0.7                       步长减小的最小因子
#define TINY 1.0e-30                     防止被零除
#define SCALMX 0.1                       1/SCALMX 是步长能够增加的最大因子
float **d,*x;                            指针指向由 pzextr 或 rzextr 调用的矩阵和向量

void bsstep(float y[], float dydx[], int nv, float *xx, float htry, float eps, float yscal[], float *hdid,
    float *hnext, void (*derivs)(float, float [], float []))
```

布里斯奇-斯托步，它用监视局部截尾误差来保证精度和调整步长。输入是自变量 x 的起始值处的因变向量 y[1..nv]及其导数 dydx[1..nv]。输入还有尝试的步长 htry，所需精度 eps，测度误差的向量 yscal[1..nv]。在输出中，y 和 x 由其新值替换，hdid 是实际采用的步长值，hnext 是步长的下一步估算值。derivs 是用户提供的程序，用以计算等式右端的导数。注意，在成功的步时，把从前面步返回的 hnext 赋给下一连续步的 htry，这样以备此程序被 odeint 调用。

```
{
    void mmid(float y[], float dydx[], int nvar, float xs, float htot,
        int nstep, float yout[], void (*derivs)(float, float[], float[]));
    void pzextr(int iest, float xest, float yest[], float yz[], float dy[],
        int nv);
    int i,iq,k,kk,km;
    static int first=1,kmax,kopt;
    static float epsold = -1.0,xnew;
    float eps1,errmax,fact,h,red,scale,work,wrkmin,xest;
    float *err,*yerr,*ysav,*yseq;
    static float a[IMAXX+1];
    static float alf[KMAXX+1][KMAXX+1];
    static int nseq[IMAXX+1]={0,2,4,6,8,10,12,14,16,18};
    int reduct,exitflag=0;

    d=matrix(1,KMAXX,1,KMAXX);
    err=vector(1,KMAXX);
    x=vector(1,KMAXX);
    yerr=vector(1,nv);
    ysav=vector(1,nv);
    yseq=vector(1,nv);
    if (eps != epsold) {                     新的容差，所以重新初始化
        *hnext = xnew = -1.0e29;             "不可能"的值
        eps1=SAFE1*eps;
        a[1]=nseq[1]+1;                      计算有效系数 A_k
        for (k=1;k<=KMAXX;k++) a[k+1]=a[k]+nseq[k+1];
        for (iq=2;iq<=KMAXX;iq++) {              计算 α(k,q).
            for (k=1;k<iq;k++)
                alf[k][iq]=pow(eps1,(a[k+1]-a[iq+1])/
                    ((a[iq+1]-a[1]+1.0)*(2*k+1)));
        }
        epsold=eps;
        for (kopt=2;kopt<KMAXX;kopt++)       确定收敛的最优行数
            if (a[kopt+1] > a[kopt]*alf[kopt-1][kopt]) break;
        kmax=kopt;
    }
    h=htry;
    for (i=1;i<=nv;i++) ysav[i]=y[i];        存储起始值
    if (*xx != xnew || h != (*hnext)) {      新的步长或新的积分：
        first=1;                                 重新建立阶次窗口
        kopt=kmax;
    }
    reduct=0;
    for (;;) {
        for (k=1;k<=kmax;k++) {              估算修改中点积分序列的值
            xnew=(*xx)+h;
            if (xnew == (*xx)) nrerror("step size underflow in bsstep");
            mmid(ysav,dydx,nv,*xx,h,nseq[k],yseq,derivs);
            xest=SQR(h/nseq[k]);             平方，因为误差级数是偶数
```

```
        pzextr(k,xest,yseq,y,yerr,nv);                执行外推
        if (k != 1) {                                 计算归一化误差估计值
            errmax=TINY;                                  ε(k).
            for (i=1;i<=nv;i++) errmax=FMAX(errmax,fabs(yerr[i]/yscal[i]));
            errmax /= eps;                            相对于容限放缩误差
            km=k-1;
            err[km]=pow(errmax/SAFE1,1.0/(2*km+1));
        }
        if (k != 1 && (k >= kopt-1 || first)) {       在阶次窗口中
            if (errmax < 1.0) {                       收敛
                exitflag=1;
                break;
            }
            if (k == kmax || k == kopt+1) {           检查可能的步长减小
                red=SAFE2/err[km];
                break;
            }
            else if (k == kopt && alf[kopt-1][kopt] < err[km]) {
                    red=1.0/err[km];
                    break;
                }
            else if (kopt == kmax && alf[km][kmax-1] < err[km]) {
                    red=alf[km][kmax-1]*SAFE2/err[km];
                    break;
                }
            else if (alf[km][kopt] < err[km]) {
                red=alf[km][kopt-1]/err[km];
                break;
            }
        }
    }
    if (exitflag) break;
    red=FMIN(red,REDMIN);                             由最小值REDMIN和最大值REDMAX
    red=FMAX(red,REDMAX);                                 减小步长
    h *= red;
    reduct=1;
}                                                     再试
*xx=xnew;                                             成功完成一步
*hdid=h;
first=0;
wrkmin=1.0e35;                                        为收敛计算最优行和相应步长
for (kk=1;kk<=km;kk++) {
    fact=FMAX(err[kk],SCALMX);
    work=fact*a[kk+1];
    if (work < wrkmin) {
        scale=fact;
        wrkmin=work;
        kopt=kk+1;
    }
}
*hnext=h/scale;
if (kopt >= k && kopt != kmax && !reduct) {
对可能出现的阶次增长进行检测若步长正好减小则不检测
    fact=FMAX(scale/alf[kopt-1][kopt],SCALMX);
    if (a[kopt+1]*fact <= wrkmin) {
        *hnext=h/fact;
        kopt++;
    }
}
free_vector(yseq,1,nv);
free_vector(ysav,1,nv);
free_vector(yerr,1,nv);
free_vector(x,1,KMAXX);
free_vector(err,1,KMAXX);
free_matrix(d,1,KMAXX,1,KMAXX);
}
```

多项式外推的程序基于和第3.9节 **polint** 中相同的算法。它是比较简单的，因为总是外推到零，而非外推到任意值。但它也是较复杂的，因为必须单独地对某数值向量中的每一分量进行外推。

```
#include "nrutil.h"

extern float **d, *x;                    在程序 bsstep 中定义

void pzextr(int iest, float yest[], float yz[], float dy[], int nv)
    使用多项式外推来计算 nv 个函数在 x=0 处的值，它是通过将多项式拟合成一列具有逐次较小值 x=xest 和相应的函数向量 yes[1..nv] 的序列来计算的。这次是调用序列中的第 iest 次调用。所外推的函数值以 yz[1..nv] 输出，它们的估计值误差输出为 dy[1..nv]。
{
    int k1,j;
    float q,f2,f1,delta,*c;

    c=vector(1,nv);
    x[iest]=xest;                          保存当前自变量
    for (j=1;j<=nv;j++) dy[j]=yz[j]=yest[j];
    if (iest == 1) {                       存储第一列估计值
        for (j=1;j<=nv;j++) d[j][1]=yest[j];
    } else {
        for (j=1;j<=nv;j++) c[j]=yest[j];
        for (k1=1;k1<iest;k1++) {
            delta=1.0/(x[iest-k1]-xest);
            f1=xest*delta;
            f2=x[iest-k1]*delta;
            for (j=1;j<=nv;j++) {          将表 1 的对角元延伸得更多
                q=d[j][k1];
                d[j][k1]=dy[j];
                delta=c[j]-q;
                dy[j]=f1*delta;
                c[j]=f2*delta;
                yz[j] += dy[j];
            }
        }
        for (j=1;j<=nv;j++) d[j][iest]=dy[j];
    }
    free_vector(c,1,nv);
}
```

目前人们在布里斯奇-斯托法中用多项式外推，而不用有理函数外推。但是，我们的感觉是，这种观点是由各种问题的测试中得出的，并非是某一种方法确实“更好些”。相应地，我们也提供了有理函数外推的最优程序 **rzextr**，它完全可替代上面的 **pzextr** 程序。

```
#include "nrutil.h"

extern float **d, *x;                    在程序 bsstep 中定义

void rzextr(int iest, float xest, float yest[], float yz[], float dy[], int nv)
    本程序可完全替代程序 pzextr，采用对角有理函数外推替换多项式外推。
{
    int k,j;
    float yy,v,ddy,c,b1,b,*fx;

    fx=vector(1,iest);
    x[iest]=xest;                          保存当前自变量
    if (iest == 1)
        for (j=1;j<=nv;j++) {
            yz[j]=yest[j];
```

```
			d[j][1]=yest[j];
			dy[j]=yest[j];
		}
	else {
		for (k=1;k<iest;k++)
			fx[k+1]=x[iest-k]/xest;
		for (j=1;j<=nv;j++) {          求表中下一对角元的值
			v=d[j][1];
			d[j][1]=yy=c=yest[j];
			for (k=2;k<=iest;k++) {
				b1=fx[k]*v;
				b=b1-c;
				if (b) {
					b=(c-v)/b;
					ddy=c*b;
					c=b1*b;
				} else                 当心，必须避免零作除数
					ddy=v;
				if (k != iest) v=d[j][k];
				d[j][k]=ddy;
				yy += ddy;
			}
			dy[j]=ddy;
			yz[j]=yy
		}
	}
	free_vector(fx,1,iest);
}
```

参考文献和进一步读物：


Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Sringer-Verlag), §7.2.14[1]

Deuflhard, P. 1983, *Numeerische Mathematik*, vol.41, pp.3??~422. [2]

Deuflhard, P. 1985, *SIAM Review*, vol.27, pp.505~535. [3]


## 16.5 二阶守恒方程组

通常情况下，当要解一组高阶微分方程组时，最好把它化成一个一阶微分方程组，如第16节中讨论的那样。但实践中通常有一类特殊的方程组，通过直接差分而效率可提高两倍。这一类方程组是二阶系统，其在等式右端没有出现导数：

$$y'' = f(x,y) \qquad y(x_0) = y_0, \qquad y'(x_0) = z_0 \tag{16.5.1}$$

和以往一样，这里的 $y$ 可以代表一个数值的向量。

分解这样的问题，最通用的方法可追溯到1907年提出的 Stoermer 法则。设 $h=H/m$，我们有

$$\begin{aligned} y_1 &= y_0 + h[z_0 + \frac{1}{2}hf(x_0,y_0)] \\ y_{k+1} - 2y_k + y_{k-1} &= h^2 f(x_0 + kh, y_k), \qquad k = 1,\ldots,m-1 \\ z_m &= (y_m - y_{m-1})/h + \frac{1}{2}hf(x_0 + H, y_m) \end{aligned} \tag{16.5.2}$$

这里，$z_m$ 是 $y'(x_0+H)$。Henrici 说明了如何通过利用值 $\Delta_k \equiv y_{k+1} - y_k$ 重写方程(16.5.2)来减小舍入误差。

首先

$$\begin{aligned} \Delta_0 &= h[z_0 + \frac{1}{2}hf(x_0,y_0)] \\ y_1 &= y_0 + \Delta_0 \end{aligned} \tag{16.5.3}$$

然后对于 $k=1,\ldots,m-1$,设置

$$
\begin{aligned}
\Delta_k &= \Delta_{k-1} + h^2 f(x_0 + kh, y_k) \\
y_{k+1} &= y_k + \Delta_k
\end{aligned}
\qquad (16.5.4)
$$

最后从下式计算导数

$$
z_m = \Delta_{m-1}/h + \frac{1}{2} h f(x_0 + H, y_m) \qquad (16.5.5)
$$

Gragg 重新说明了方程组(16.5.3)~(16.5.5)包含的系列误差只含有 h 有偶数次方,这样,这种方法逻辑上便可适用于布里斯奇-斯托法,我们用下列程序 **stoerm** 代替 **mmid**:

```
#include "nrutil.h"

void stoerm (float y[], float d2y[], int nv, float xs, float htot, int nstep,
    float yout[],void ( * derivs)(float, float [], float []))
    对于n=nv/2个方程,用Stoermer法则求解y''=f(x,y)的程序。一个输入y[1..nv]在前n个元素中包含y,在后n
    个元素中包含在xs处的y'值,d2y[1..nv]前n个元素包含右边的函数值f(也是在xs处),后n个元素没有用到,另
    一输入量htot,总共要采取的步数,及nstep,要采用的子步数,输出返回在yout[1..nv]中,和y的存储结构一样。
    derivs是用户提供用于计算f的程序。
{
    int i,n,neqns,nn;
    float h,h2,halfh,x,*ytemp;

    ytemp=vector(1,nv);
    h=htot/nstep;                        这次的步长
    halfh=0.5*h;
    neqns=nv/2;                          方程个数
    for (i=1;i<=neqns;i++) {             第一步
        n=neqns+i;
        ytemp[i]=y[i]+(ytemp[n]=h*(y[n]+halfh*d2y[i]));
    }
    x=xs+h;
    (*derivs)(x,ytemp,yout);             利用yout暂时存贮导数值
    h2=h*h;
    for (nn=2;nn<=nstep;nn++) {          一般情况
        for (i=1;i<=neqns;i++)
            ytemp[i] += (ytemp[(n=neqns+i)] += h2*yout[i]);
        x += h;
        (*derivs)(x,ytemp,yout);
    }
    for (i=1;i<=neqns;i++) {             最后一步
        n=neqns+i;
        yout[n]=ytemp[n]/h+halfh*yout[i];
        yout[i]=ytemp[i];
    }
    free_vector(ytemp,1,nv);
}
```

注意,为了和 **bsstep** 兼容,数组 y 和 d2y 的长度对 n 个二阶方程是 $2n$,y 值存储在 y 的前 n 个元素中,而后 n 个元素中存储第一阶求导值。等式右边的 f 存储在 d2y 矩阵的前 n 个元素中,后 n 个元素未使用。在这样的存储结构下,可以使用 bsstep,只需简单地用同样的参数值调用 **stoerm** 来代替对 **mmid** 的调用;只是要注意 **bsstep** 的参数 nv 设置成 2n,也应使用 Deuflhand 建议的更有效的步长序列:

$$
n = 1,2,3,4,5,\ldots \qquad (16.5.6)
$$

并在 **bsstep** 中将 KMAXX 设置成 12 。

## 参考文献及进一步读物:


Devflhard, P. 1985, SIAM *Review*, vol. 27, pp. 505~535.

## 16.6 方程的刚性集

当处理多于一个的一阶微分方程时，方程的刚性集也许就可能出现了。刚性问题通常出现在自变量的取值有两个或多个不同尺度，而应变量正随此而变化的问题中。例如，考虑下面的方程组

$$\begin{aligned} u' &= 998u + 1998v \\ v' &= -999u - 1999v \end{aligned} \tag{16.6.1}$$

其边界值条件为

$$u(0) = 1 \qquad v(0) = 0 \tag{16.6.2}$$

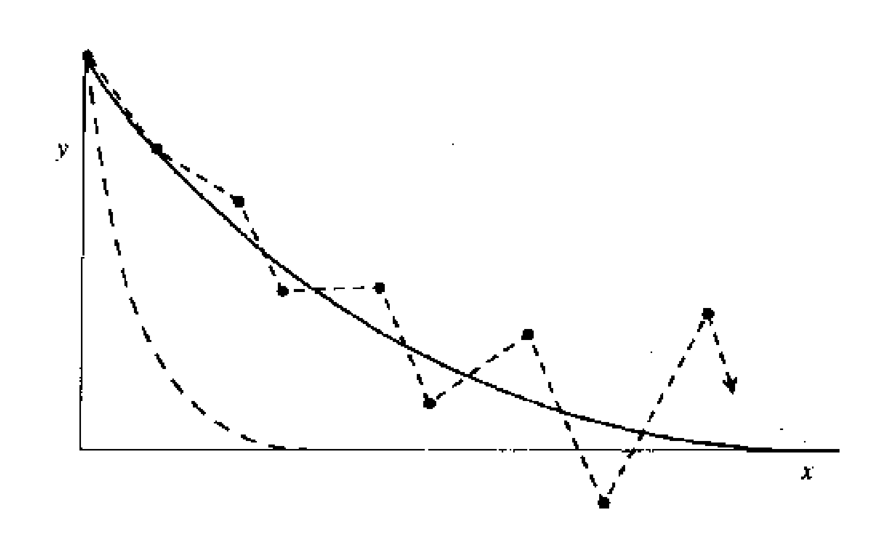
通过变形

$$u = 2y - z \qquad v = -y + z \tag{16.6.3}$$

我们得到解

$$\begin{aligned} u &= 2e^{-x} - e^{-1000x} \\ v &= -e^{-x} + e^{-1000x} \end{aligned} \tag{16.6.4}$$

如果我们用这一章内到目前为止的任意一种方法，对(15.6.1)方程组进行积分求解，其中的 $e^{-1000x}$ 项需要步长 $h \ll 1/1000$ 才能使方法稳定（其原因在下面介绍）。即使当 $x$ 离开原点，在决定 $u$ 和 $v$ 值的过程中 $e^{-1000x}$ 项完全可以忽略时也得这样（见图16.6.1）




这里假定方程有两组解，如上面实线和短横线所示。尽管初始条件是提供满足实线解的，但积分的稳定性（如图中不稳定的带点的线段序列）则是由变化很快的短横线解所决定的，即使在那个解几乎趋近于零时也一样。要解决这个问题可以利用隐式积分。

图16.6.1 积分刚性方程中遇到不稳定性的例子（图示）

这是刚性方程的通病：我们需要在解的取值较小的度量范围内变化步长，以保持积分的稳定性，尽管积分精度可以允许一个比较大的步长值。

我们来看一下怎么解决这个问题。考虑下面这个方程：

$$y' = -cy \tag{16.6.5}$$

这里 $c>0$ 是一个常数。用隐式(或前向)欧拉法以步长 $h$ 积分，这个方程是

$$y_{n+1} = y_n + hy'_n = (1 - ch)y_n \tag{16.6.6}$$

这种方法称作隐式解法，因为新的值 $y_{n+1}$ 是以旧的值 $y_n$ 的隐含方式给出的。显然，当 $h>2/c$ 时，这种方法是不稳定的，因为当 $n\to\infty$ 时，$y_n\to\infty$。

最简便的解决方法是采取**隐式差分**，其中在新的 $y$ 值处估算右端项。在这种情况下，我们得到了**后向欧拉法**：

$$y_{n+1} = y_n + hy'_{n+1} \tag{16.6.7}$$

或

$$y_{n+1} = \frac{y_n}{1 + ch} \tag{16.6.8}$$

这种方法是绝对稳定的：即使当 $h\to\infty$ 时，亦有 $y_{n+1}\to 0$，这实际上是微分方程的正确解。如果我们把 $x$ 设想成代表时间，则隐式方法将以大的步长收敛到真实的平衡解(如时间稍晚一些的解)。这种隐式方法的良好特性只对线性系统成立，但即使在一般情况下，隐式方法也能给出比较好的稳定性。当然，如果我们用较大的步长逼近真实解时，我们得放弃精确度，但我们还是坚持稳定性优先。

这些思想能够很容易地推广到常系数线性方程组的情况：

$$\mathbf{y}' = -\mathbf{C}\cdot\mathbf{y} \tag{16.6.9}$$

其中 $\mathbf{C}$ 是一个正定矩阵。隐含差分法为：

$$\mathbf{y}_{n+1} = (\mathbf{1} - \mathbf{C}h)\cdot\mathbf{y}_n \tag{16.6.10}$$

一个矩阵 $\mathbf{A}^n$ 当 $n\to\infty$ 时趋近于零，仅当 $\mathbf{A}$ 的最大特征值小于单位 1。因此，当 $n\to\infty$ 时，$\mathbf{y}_n$ 有界的充分条件是，仅当 $\mathbf{1}-\mathbf{C}h$ 的最大特征值小于 1，换言之

$$h < \frac{2}{\lambda_{\max}} \tag{16.6.11}$$

这里，$\lambda_{\max}$ 是 $\mathbf{C}$ 的最大特征值。

另一方面，隐含差分式给出了

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{y}'_{n+1} \tag{16.6.12}$$

或

$$\mathbf{y}_{n+1} = (\mathbf{1} + \mathbf{C}h)^{-1}\cdot\mathbf{y}_n \tag{16.6.13}$$

如果 $\mathbf{C}$ 的特征值是 $\lambda$，则 $(\mathbf{1}+\mathbf{C}h)^{-1}$ 的特征值应是 $(1+\lambda h)^{-1}$，对所有 $h$ 来说，它的量值都小于1(正定矩阵的所有特征值都是非负的)。因此这种方法对于所有的步长 $h$ 都是稳定的。为达到这种稳定性，我们所付出的代价是每一步都对矩阵求逆。

遗憾的是，并非所有方程都是常系数的线性方程。对于诸如方程组

$$\mathbf{y}' = \mathbf{f}(\mathbf{y}) \tag{16.6.14}$$

隐式差分法导出：

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{f}(\mathbf{y}_{n+1}) \tag{16.6.15}$$

一般说来，这是一个比较麻烦的非线性方程组，必须每步进行迭代处理。假设我们能化成线性方程组，象牛顿法中那样：

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\left[\mathbf{f}(\mathbf{y}_n) + \left.\frac{\partial\mathbf{f}}{\partial\mathbf{y}}\right|_{\mathbf{y}_n}\cdot(\mathbf{y}_{n+1} - \mathbf{y}_n)\right] \tag{16.6.16}$$

其中 $\partial\mathbf{f}/\partial\mathbf{y}$ 是右边矩阵(雅可比矩阵)的偏导数。重新将方程(16.6.16)安排一下，有

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\left[\mathbf{1} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right]^{-1} \cdot \mathbf{f}(\mathbf{y}_n) \tag{16.6.17}$$

如果 $h$ 不是很大，也许只用一次牛顿法迭代就足以精确地用式(16.6.1)求解方程(16.6.15)。也就是说，在每一步，我们必须把矩阵

$$\mathbf{1} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}} \tag{16.6.18}$$

求逆以求出 $\mathbf{y}_{n+1}$。用线性化解决隐式解法的问题称为“半隐式解法”。这样，等式(16.6.17)便是**半隐式欧拉方法**。这种方法并不保证是稳定的，但通常是稳定的，因为其局部表现很类似于上面所述的常数矩阵 **C** 的情况。

这样，我们讨论了有一阶精确度的隐式算法，这些算法可用性是很强的。大多数问题可从高阶算法中得到满意结果。下面是刚性方程组三种重要的高阶算法。

- 通用的龙格-库塔法，其中最有用的是罗塞布鲁克(Rosenbock)算法。首次用该思想进行实践的是 Kaps 和 Retrop，所以这些算法又被称为 Kaps-Rentrop 算法。
- 通用的布里斯奇-斯托算法，尤其是 Bader 和 Deuflhand 提出的半稳式外推算法。
- 预测校正法，其主要部分是 Gear 后向差分法的继续。

我们将给出前两种算法的实现。注意，如果方程的右边显式地依赖于 $x$、$\mathbf{f}(\mathbf{y},x)$，可以通过把 $x$ 加到应变量的列表中来解决。这样，方程组变成

$$\begin{pmatrix} \mathbf{y} \\ x \end{pmatrix}' = \begin{pmatrix} \mathbf{f} \\ 1 \end{pmatrix} \tag{16.6.19}$$

在本节要给出的两个程序中，我们已把这种替换显式地完成了。这样，用户就不用再为考虑 $\mathbf{f}(\mathbf{y},x)$ 而费多大劲了。

现在，我们提醒很重要的一点：在用自动调整步长来积分刚性方程时，适当地标度变元是绝对很关键的事。和非刚性方程求解程序一样，需提供一个向量 $\mathbf{y}_{scal}$，用其来测量误差。例如，为得到常数部分误差，只需简单地设置 $\mathbf{y}_{scal} = |\mathbf{y}|$。也可以通过把 $\mathbf{y}_{scal}$ 设置成其最大值，来得到常量相对于某一最大值的绝对误差。在刚性方程中，在其解中常常有递减极其迅速的部分，当其变得很小时，用户可能不十分感兴趣。这时，可以控制高于某一阈值 **C** 的相对误差、或是低于该阈值的绝对误差，通过设置

$$\mathbf{y}_{scal} = \max(\mathbf{C}, |\mathbf{y}|) \tag{16.6.20}$$

如果用某一适当的无量纲的单位，则 **C** 的每一分量是有序的单位值。如果不能确定应该给 **C** 采用什么样的值，则只需试着把每个分量设成和单位值相同即可。对于刚性问题，我们积极提倡选择式(16.6.20)。

最后警告一点：解决刚性问题有时可能导致精度的极大差距。所以要注意，应使用双精度型的情况。

### 16.6.1 罗塞布鲁克算法

这些算法在理解和实现起来相对比较简单。对于中等精度要求($\epsilon \leqslant 10^{-4} \sim 10^{-5}$)以及中等复杂的问题($N \leqslant 10$)，它们和更加复杂的算法是可相比拟的。对于更加严格的参数，罗塞布鲁克(Rosenbrook)算法仍然有效；这些算法在有效性上只是比半隐含外推算法(见后面)要差一些。

罗塞布鲁布算法寻找了一个形式如下的解

$$\mathbf{y}(x_0 + h) = \mathbf{y}_0 + \sum_{i=1}^{s} c_i \mathbf{k}_i \tag{16.6.21}$$

其中校正系数 $\mathbf{k}_i$ 通过综合式(16.6.17)中结构的 $s$ 个线性方程进行求解而获得：

$$(\mathbf{1} - \gamma h \mathbf{f}') \cdot \mathbf{k}_i = h\mathbf{f}\left(\mathbf{y}_0 + \sum_{j=1}^{i-1} \alpha_{ij} \mathbf{k}_j\right) + h\mathbf{f}' \cdot \sum_{j=1}^{i-1} \gamma_{ij} \mathbf{k}_j, \qquad i = 1, \dots, s \tag{16.6.22}$$

其中，用 $\mathbf{f}'$ 表示雅可比矩阵，系数 $\gamma$、$c_i$、$\alpha_{ij}$ 和 $\gamma_{ij}$ 是和问题无关的固定参数。如果 $\gamma = \gamma_{ij} = 0$，这是简单的龙格-库塔法，方程(16.6.22)可被成功地解出 $\mathbf{k}_1, \mathbf{k}_2, \dots$。

求解刚性方程组成功的关键是，步长的自动调整算法。Kaps 和 Rentrop[2]发现了一种嵌入式或龙格-库塔-Fechlkceg 算法。如第16.2节中所述：计算式(16.6.21)的两个估计值、"真实值" $\mathbf{y}$ 以及有不同的系数 $\hat{c}_i$ $(i = 1, \dots, \hat{s})$ 的低阶估计值，这里 $\hat{s} < s$，但 $\mathbf{k}_i$ 是相同的。$\mathbf{y}$ 和 $\hat{\mathbf{y}}$ 之差可以导出局部截断误差的估计值，这可以用于步长控制。Kaps 和 Rentrop 指出，取嵌入的最小 $s$ 可能是 $s = 4$，$\hat{s} = 3$，从而成为一个四阶算法。

为使式(16.6.22)右端的矩阵向量的乘积最小，我们用下列量重写方程：

$$\mathbf{g}_i = \sum_{j=1}^{i-1} \gamma_{ij} \mathbf{k}_j + \gamma \mathbf{k}_i \tag{16.6.23}$$

则方程变成下面形式

$$\begin{aligned}
(1/\gamma h - \mathbf{f}') \cdot \mathbf{g}_1 &= \mathbf{f}(\mathbf{y}_0) \\
(1/\gamma h - \mathbf{f}') \cdot \mathbf{g}_2 &= \mathbf{f}(\mathbf{y}_0 + a_{21}\mathbf{g}_1) + c_{21}\mathbf{g}_1/h \\
(1/\gamma h - \mathbf{f}') \cdot \mathbf{g}_3 &= \mathbf{f}(\mathbf{y}_0 + a_{31}\mathbf{g}_1 + a_{32}\mathbf{g}_2) + (c_{31}\mathbf{g}_1 + c_{32}\mathbf{g}_2)/h \\
(1/\gamma h - \mathbf{f}') \cdot \mathbf{g}_4 &= \mathbf{f}(\mathbf{y}_0 + a_{41}\mathbf{g}_1 + a_{42}\mathbf{g}_2 + a_{43}\mathbf{g}_3) + (c_{41}\mathbf{g}_1 + c_{42}\mathbf{g}_2 + c_{43}\mathbf{g}_3)/h
\end{aligned} \tag{16.6.24}$$

在我们根据 Kaps-Rentrop 算法实施的程序中，我们已经在方程(16.6.24)中显式地实现了式(16.6.19)中的代换，所以用户自己不用为此担心了，只需简单地提供一个程序(在 stiff 中称为 derivs)，该程序返回 $\mathbf{f}$(称作 dydx)作为 $x$ 和 $\mathbf{y}$ 的函数，同时也提供一个程序 jacobn，该程序返回 $\mathbf{f}'$(dfdy)以及 $\partial \mathbf{f}/\partial x$(dfdx)作为 $x$ 和 $\mathbf{y}$ 的函数。如果 $x$ 在右端没有显式出现，则 dfdx 为零。通常雅可比矩阵可通过对右端 $\mathbf{f}$ 解析微分而得到。如果不能，用户的程序必须用适当的增量 $\Delta \mathbf{y}$ 通过数值差来计算。

Kaps 和 Rentrop 给出了两组不同的参数集，在稳定性方面有稍许的不同。另外还提出了其它几组参数。我们缺省的选择是 Shampine[3]提供的参数，但也给出其中一组 Kaps-Rentrop 参数作为可选项。有些提出的参数集需要有在积分域外的函数估值，我们希望避开那些复杂度。

调用 **stiff** 的参数序列和早先给出的非刚性程序中的一样，因此它和一般常微分方程组中所给程序 **odeint** 是兼容的。遗憾的是，这种兼容性需要有一些稍微不同：用户提供的程序 **derivs** 是一个哑元(无实际名字)，而另一个用户提供的程序不是一个参数，而必须命名为 **jacobn**。

**stiff** 程序首先存储初始值，以备在误差超过时该步能重新计算。线性方程组(16.6.24)首先用程序 **lubcmp** 对矩阵 $\mathbf{1}/\gamma h - \mathbf{f}'$ 进行 $LU$ 分解，然后用 **lubksb** 对四个不同的右端项进行回代求出 $\mathbf{g}_i$。注意，积分的每一步需要调用 **jacobn** 一次，调用 **derivs** 三次(是在 **stiff** 之前调用一次获取 dydx，并在 **stiff** 内部调用两次)。为什么只需三次而非四次调用，原因是方程组(16.6.24)中最后的两个调用参数已经被选择，所以只需用相同的参数即可。雅可比矩阵的计算量大致等同于对右端 $\mathbf{f}$ 的 $N$ 个估值的计算量，我们看出 Kaps-Rentrop 方法每步要进行 $N+3$ 次函数的估算。注意，如果 $N$ 较大，而雅可比矩阵是稀疏矩阵，就应该用一合适的稀疏矩阵算法来代替 $LU$ 分解。

步长控制依赖于

$$\begin{aligned}
\mathbf{y}_{\text{exact}} &= \mathbf{y} + O(h^5) \\
\mathbf{y}_{\text{exact}} &= \hat{\mathbf{y}} + O(h^4)
\end{aligned} \tag{16.6.25}$$

因此

$$|\mathbf{y} - \hat{\mathbf{y}}| = O(h^4) \tag{16.6.26}$$

参考方程(16.2.4)到(16.2.10)的起始步长,我们看到新的步长应该选为如式(16.2.10),但是把指数 1/4 和 1/5 相应地用 1/3 和 1/4 代替。同样,经验表明在一步计算中,应避免步长发生大的变化,否则我们可能要在下一步中取消这种大的变化。我们采用 0.5 和 1.5 作为一步中允许 $h$ 的最大减量和增量。

```
#include <math.h>
#include "nrutil.h"
#define SAFETY 0.9
#define GROW 1.5
#define PGROW   0.25
#define SHRNK 0.5
#define PSHRNK (-1.0/3.0)
#define ERRCON 0.1296
#define MAXTRY 40
    这里 NMAX 是 n 的最大值;GROW 和 SHRNK 是在一步中步长能变化的最大和最小因子;ERRCON 等于
    (GROW/SAFETY)的(1/PGROW)次幂,并处理 errmax≃0的情况
#define GAM (1.0/2.0)
#define A21 2.0
#define A31 (48.0/25.0)
#define A32 (6.0/25.0)
#define C21 -8.0
#define C31 (372.0/25.0)
#define C32 (12.0/5.0)
#define C41 (-112.0/125.0)
#define C42 (-54.0/125.0)
#define C43 (-2.0/5.0)
#define B1 (19.0/9.0)
#define B2 (1.0/2.0)
#define B3 (25.0/108.0)
#define B4 (125.0/108.0)
#define E1 (17.0/54.0)
#define E2 (7.0/36.0)
#define E3 0.0
#define E4 (125.0/108.0)
#define C1X (1.0/2.0)
#define C2X (-3.0/2.0)
#define C3X (121.0/50.0)
#define C4X (29.0/250.0)
#define A2X 1.0
#define A3X (3.0/5.0)

void stiff(float y[], float dydx[], int n, float *x, float htry, float eps,float yscal[], float *hdid,
    float *hnext,void (*derivs)(float, float [], float []))
    四阶罗塞布鲁克求解刚度常微分方程,用监视局部截尾误差以调整步长。输入是自变量 x 的初始值处的因变量 y
    [1..n]及其导数 dydx[1..n]。输入还有尝试的步长 htry,所需精度 eps,以及量测误差的向量 yscal[1..n]。在输出
    中,y 和 x 由其新值替换,hdid 是实际采用的步长值,hnext 是步长的下一步估算值。derivs 是用户提供的程序,用以
    计算等式右边相应于 x 值。而 jacobn(固定的名字)是用户提供的程序,用以计算相应 y 元素的右端项导数的雅可比
    矩阵。
{
    void jacobn(float x, float y[], float dfdx[], float **dfdy, int n);
    void lubksb(float **a, int n, int *indx, float b[]);
    void ludcmp(float **a, int n, int *indx, float *d);
    int i,j,jtry,*indx;
    float d,errmax,h,xsav,**a,*dfdx,**dfdy,*dysav,*err;
    float *g1,*g2,*g3,*g4,*ysav;

    indx=ivector(1,n);
    a=matrix(1,n,1,n);
    dfdx=vector(1,n);
```

```
dfdy=matrix(1,n,1,n);
dysav=vector(1,n);
err=vector(1,n);
g1=vector(1,n);
g2=vector(1,n);
g3=vector(1,n);
g4=vector(1,n);
ysav=vector(1,n);
xsav=(*x);                          保存初始值
for (i=1;i<=n;i++) {
    ysav[i]=y[i];
    dysav[i]=dydx[i];
}
jacobn(xsav,ysav,dfdx,dfdy,n);
用户提供的程序返回n×n阶矩阵dfdy和向量dfdx
h=htry;                             给初始判决值设置步长
for (jtry=1;jtry<=MAXTRY;jtry++) {
    for (i=1;i<=n;i++) {            建立矩阵1 − γhf'
        for (j=1;j<=n;j++) a[i][j] = -dfdy[i][j];
        a[i][i] += 1.0/(GAM*h);
    }
    ludcmp(a,n,indx,&d);            矩阵LU分解
    for (i=1;i<=n;i++)              为g1设置右边
        g1[i]=dysav[i]+h*C1X*dfdx[i];
    lubksb(a,n,indx,g1);            解g1
    for (i=1;i<=n;i++)              计算中间变元y和x
        y[i]=ysav[i]+A21*g1[i];
    *x=xsav+A2X*h;
    (*derivs)(*x,y,dydx);           计算中间变元处的dydx
    for (i=1;i<=n;i++)              为g2设置右边
        g2[i]=dydx[i]+h*C2X*dfdx[i]+C21*g1[i]/h;
    lubksb(a,n,indx,g2);            解g2
    for (i=1;i<=n;i++)              计算中间变元y和x
        y[i]=ysav[i]+A31*g1[i]+A32*g2[i];
    *x=xsav+A3X*h;
    (*derivs)(*x,y,dydx);           计算中间值处的dydx
    for (i=1;i<=n;i++)              为g3设置右边
        g3[i]=dydx[i]+h*C3X*dfdx[i]+(C31*g1[i]+C32*g2[i])/h;
    lubksb(a,n,indx,g3);            解g3
    for (i=1;i<=n;i++)              为g4设置右边
        g4[i]=dydx[i]+h*C4X*dfdx[i]+(C41*g1[i]+C42*g2[i]+C43*g3[i])/h;
    lubksb(a,n,indx,g4);            解g4
    for (i=1;i<=n;i++) {            得到y的四阶估值及误差估值
        y[i]=ysav[i]+B1*g1[i]+B2*g2[i]+B3*g3[i]+B4*g4[i];
        err[i]=E1*g1[i]+E2*g2[i]+E3*g3[i]+E4*g4[i];
    }
    *x=xsav+h;
    if (*x == xsav) nrerror("stepsize not significant in stiff");
    errmax=0.0;                     计算精度
    for (i=1;i<=n;i++) errmax=FMAX(errmax,fabs(err[i]/yscal[i]));
    errmax /= eps;                  相对于所需误差容限放缩
    if (errmax <= 1.0) {            该步成功，计算下一步长并返回
        *hdid=h;
        *hnext=(errmax > ERRCON ? SAFETY*h*pow(errmax,PGROW) : GROW*h);
        free_vector(ysav,1,n);
        free_vector(g4,1,n);
        free_vector(g3,1,n);
        free_vector(g2,1,n);
        free_vector(g1,1,n);
        free_vector(err,1,n);
        free_vector(dysav,1,n);
        free_matrix(dfdy,1,n,1,n);
        free_vector(dfdx,1,n);
        free_matrix(a,1,n,1,n);
        free_ivector(indx,1,n);
        return;
    } else {                        截断误差太大，减小步长！
```

```
            *hnext=SAFETY*h*pow(errmax,PSHRNK);
            h=(h >= 0.0 ? FMAX(*hnext,SHRNK*h) : FMIN(*hnext,SHRNK*h));
        }
    }                                    回头重试
    nrerror("exceeded MAXTRY in stiff");
}
```

只需在程序中替换掉#define 语句，就可简单地将其中 Kaps Rentrop 参数替换成 Shampine。

```
#define GAM 0.231
#define A21 2.0
#define A31 4.52470820736
#define A32 4.16352878860
#define C21 -5.07167533877
#define C31 6.02015272865
#define C32 0.159750684673
#define C41 -1.856343618677
#define C42 -8.50538085819
#define C43 -2.08407513602
#define B1 3.95750374663
#define B2 4.62489238836
#define B3 0.617477263873
#define B4 1.282612945268
#define E1 -2.30215540292
#define E2 -3.07363448539
#define E3 0.873280801802
#define E4 1.282612945268
#define C1X GAM
#define C2X -0.396296677520e-01
#define C3X 0.550778939579
#define C4X -0.553509845700e-01
#define A2X 0.462
#define A3X 0.880208333333
```

作为一个例子，说明 **stiff** 是怎样用的，它可求解下面问题

$$
\begin{aligned}
y_1' &= -0.013y_1 - 1000y_1y_3 \\
y_2' &= -2500y_2y_3 \\
y_3' &= -0.013y_1 - 1000y_1y_3 - 2500y_2y_3
\end{aligned}
\tag{16.6.27}
$$

具有初始值为

$$
y_1(0) = 1, \quad y_2(0) = 1, \quad y_3(0) = 0 \tag{16.6.28}
$$

(这是[4]中的测试问题 $D_4$)。我们用 **odeint** 程序，以初始步长 $h=2.9\times10^{-4}$，计算积分方程组到 $x=50$。在式(6.6.20)中的 C 的元素都设成单位值。该问题的程序 **derivs** 和 **jacobn** 如下所示。尽管最大的到最小常数系数的比值糟糕到 $10^6$左右，但 **stiff** 还是成功地仅用 29 步就求解出了这个问题，而误差为 $\epsilon=10^{-4}$。相比较而言，龙格-库塔法 **rkqs** 需要 51012 步！

```
void jacobn(float x, float y[], float dfdx[], float **dfdy, int n)
{
    int i;

    for (i=1;i<=n;i++) dfdx[i]=0.0;
    dfdy[1][1] = -0.013-1000.0*y[3];
    dfdy[1][2]=0.0;
    dfdy[1][3] = -1000.0*y[1];
    dfdy[2][1]=0.0;
    dfdy[2][2] = -2500.0*y[3];
    dfdy[2][3] = -2500.0*y[2];
```

```
    dfdy[3][1] = -0.013-1000.0*y[3];
    dfdy[3][2] = -2500.0*y[3];
    dfdy[3][3] = -1000.0*y[1]-2500.0*y[2];
}

void derivs(float x, float y[], float dydx[])
{
    dydx[1] = -0.013*y[1]-1000.0*y[1]*y[3];
    dydx[2] = -2500.0*y[2]*y[3];
    dydx[3] = -0.013*y[1]-1000.0*y[1]*y[3]-2500.0*y[2]*y[3];
}
```

## 16.6.2 半隐式外推算法

布里斯奇-斯托算法，用修正中点的方法分离微分方程。但对刚性方程无效。Bader 和 Deuflhard[ ]发现了半隐外推方法，效果很好。而且能和最初的布里斯奇-斯托法一样有效。

起始点是一个隐式中点规则形式：

$$\mathbf{y}_{n+1} - \mathbf{y}_{n-1} = 2h\mathbf{f}\left(\frac{\mathbf{y}_{n+1} + \mathbf{y}_{n-1}}{2}\right) \tag{16.6.29}$$

线性化该等式右端 $\mathbf{f}(\mathbf{y}_n)$，把方程(16.6.29)转换成半隐式，结果便是**半隐式中点规则**：

$$\left[\mathbf{1} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right]\cdot \mathbf{y}_{n+1} = \left[\mathbf{1} + h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right]\cdot \mathbf{y}_{n-1} + 2h\left[\mathbf{f}(\mathbf{y}_n) - \frac{\partial \mathbf{f}}{\partial \mathbf{y}}\cdot \mathbf{y}_n\right] \tag{16.6.30}$$

使用了特殊的第一步，半隐式欧拉步(16.6.17)，以及在最后一步特殊的"平滑"，其中最后的 $\mathbf{y}_n$ 用

$$\bar{\mathbf{y}}_n \equiv \frac{1}{2}(\mathbf{y}_{n+1} + \mathbf{y}_{n-1}) \tag{16.6.31}$$

来代换。Bader 和 Deuflhard 说明，这种方法的误差级数也仅与 $h$ 的偶次方相关。

在实际应用中，最好用 $\Delta_k \equiv \mathbf{y}_{k+1} - \mathbf{y}_k$ 改写一下方程。利用 $h = H/m$，首先计算

$$\Delta_0 = \left[\mathbf{1} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right]^{-1}\cdot h\mathbf{f}(\mathbf{y}_0)$$
$$\mathbf{y}_1 = \mathbf{y}_0 + \Delta_0 \tag{16.6.32}$$

然后对于 $k = 1, \ldots, m=1$，设置

$$\Delta_k = \Delta_{k-1} + 2\left[\mathbf{1} - \frac{h\partial \mathbf{f}}{\partial \mathbf{y}}\right]^{-1}\cdot[h\mathbf{f}(\mathbf{y}_k) - \Delta_{k-1}]$$
$$\mathbf{y}_{k+1} = \mathbf{y}_k + \Delta_k \tag{16.6.33}$$

最后计算

$$\Delta_m = \left[\mathbf{1} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right]^{-1}\cdot[h\mathbf{f}(\mathbf{y}_m) - \Delta_{m-1}]$$
$$\bar{\mathbf{y}}_m = \mathbf{y}_m + \Delta_m \tag{16.6.34}$$

在上面的公式中，很容易和式(16.6.19)替换公式结合起来。在雅可比中，来自 $\partial \mathbf{f}/\partial x$ 的额外项都抵销了半隐性中点规则(16.6.30)。在特殊的第一步(16.6.17)及相应的方程(16.6.32)中，项 $h\mathbf{f}$ 变成 $h\mathbf{f} + h^2(\partial \mathbf{f}/\partial x)$，其它方程都未变。

**该算法在程序 simpr 中实现。**

```
#include "nrutil.h"

void simpr(float y[], float dydx[], float dfdx[], float **dfdy, int n, float xs, float htot, int nstep,
    float yout[], void (*derivs)(float, float[], float[]))
    执行一步半隐式中点规则算法。输入因变量y[1..n]，导数dydx[1..n]，相对于x的右端项导数dfdx[1..n]及在xs
```

处的雅可比矩阵 dfdy[1..n][1..n]。输入还有 htot，总共要取的步数和 nstep，被用的了步数。输出返回在 yout[1..n]中，**derivs** 是用户提供的程序，用来计算 dydx。

```
{
    void lubksb(float **a, int n, int *indx, float b[]);
    void ludcmp(float **a, int n, int *indx, float *d);
    int i,j,nn,*indx;
    float d,h,x,**a,*del,*ytemp;

    indx=ivector(1,n);
    a=matrix(1,n,1,n);
    del=vector(1,n);
    ytemp=vector(1,n);
    h=htot/nstep;                          这一步的步长
    for (i=1;i<=n;i++) {                   设置矩阵 1 - hf'
        for (j=1;j<=n;j++) a[i][j] = -h*dfdy[i][j];
        ++a[i][i];
    }
    ludcmp(a,n,indx,&d);                   矩阵LU分解
    for (i=1;i<=n;i++)                     为第一步建立右端项，用yout 作暂时存储器
        yout[i]=h*(dydx[i]+h*dfdx[i]);
    lubksb(a,n,indx,yout);
    for (i=1;i<=n;i++)                     第一步
        ytemp[i]=y[i]+(del[i]=yout[i]);
    x=xs+h;
    (*derivs)(x,ytemp,yout);               用yout 暂存导数值
    for (nn=2;nn<=nstep;nn++) {
        for (i=1;i<=n;i++)                 为一般步设右端项
            yout[i]=h*yout[i]-del[i];
        lubksb(a,n,indx,yout);
        for (i=1;i<=n;i++)
            ytemp[i] += (del[i] += 2.0*yout[i]);
        x += h;
        (*derivs)(x,ytemp,yout);
    }
    for (i=1;i<=n;i++)                     为最后一步设右端项
        yout[i]=h*yout[i]-del[i];
    lubksb(a,n,indx,yout);
    for (i=1;i<=n;i++)                     最后一步
        yout[i] += ytemp[i];
    free_vector(ytemp,1,n);
    free_vector(del,1,n);
    free_matrix(a,1,n,1,n);
    free_ivector(indx,1,n);
}
```

程序 **simpr** 可以用于程序 **stifbs** 中，基本上和 **bsstep** 一样，不同的是：

· 步长的序列是

$$n = 2,6,10,14,22,34,50,\ldots, \tag{16.6.35}$$

其中每值和前面一个值相差 4 的倍数，使得相邻项的比值 ≤5/7，参数 KMAXX 设为 7 。

· 每一步的工作量，不但包括函数求值的工作量，还包括雅可比计算的计算量。我们计算雅可比的工作量和 $N$ 个函数的估计值相当，这里 $N$ 是方程的数目。

· 用户提供的程序 **derivs** 是哑元，所以能取任何名字。但是，如果要和 **rkqs**、**bsstep** 及 **stiff** 保持最简单的兼容，**jacobn** 就不是一个参数，而必须定为这个名字。每一步被调用后返回作为 x 和 y 的函数 f'(dfdx)以及 $\partial f/\partial x$(dfdx)。

下面程序，注解中说明了和 **bsstep** 的不同之处：

```
#include <math.h>
#include "nrutil.h"
#define KMAXX 7
#define IMAXX (KMAXX+1)
```

```
#define SAFE1 0.25
#define SAFE2 0.7
#define REDMAX 1.0e-5
#define REDMIN 0.7
#define TINY 1.0e-30
#define SCALMX 0.1

float **d, *x;

void stifbs(float y[], float dydx[], int nv, float *xx, float htry, float eps, float yscal[], float *hdid,
    float *hnext, void (*derivs)(float, float [], float []))
```

半隐式外推法求解刚度微分方程。用监视局部截断误差以调整步长。输入是自变量 x 起始点处的因变量 y[1..n] 及其导数值 dydx[1..n]，输入还有试用的步长 htry、所需精度 eps 以及向量 yscal[1..n]通过该项来标度误差. 在输出中，y 和 x 被其新值所替代，hdid 是实际采用的步长，hnext 是估算的下一个步长，**derivs** 是用户提供的程序计算右端项相对于 $x$ 的导数，而 **jacobn**(名字固定)是用户提供的程序，计算右端项相对于 $y$ 元素的导数的雅可比矩阵。注意，在成功的步时，把从前面步返回的 hnext 赋给 htry 这样以备此程序被 **odeint** 调用。

```
{
    void jacobn(float x, float y[], float dfdx[], float **dfdy, int n);
    void simpr(float y[], float dydx[], float dfdx[], float **dfdy,
        int n, float xs, float htot, int nstep, float yout[],
        void (*derivs)(float, float [], float []));
    void pzextr(int iest, float xest, float yest[], float yz[], float dy[],
        int nv);
    int i,iq,k,kk,km;
    static int first=1,kmax,kopt,nvold = -1;
    static float epsold = -1.0,xnew;
    float eps1,errmax,fact,h,red,scale,work,wrkmin,xest;
    float *dfdx,**dfdy,*err,*yerr,*ysav,*yseq;
    static float a[IMAXX+1];
    static float alf[KMAXX+1][KMAXX+1];
    static int nseq[IMAXX+1]={0,2,6,10,14,22,34,50,70};      序列和bsstep不同
    int reduct,exitflag=0;

    d=matrix(1,KMAXX,1,KMAXX);
    dfdx=vector(1,nv);
    dfdy=matrix(1,nv,1,nv);
    err=vector(1,KMAXX);
    x=vector(1,KMAXX);
    yerr=vector(1,nv);
    ysav=vector(1,nv);
    yseq=vector(1,nv);
    if(eps != epsold || nv != nvold) {          如果nv改变则重新初始化
        *hnext = xnew = -1.0e29;
        eps1=SAFE1*eps;
        a[1]=nseq[1]+1;
        for (k=1;k<=KMAXX;k++) a[k+1]=a[k]+nseq[k+1];
        for (iq=2;iq<=KMAXX;iq++) {
            for (k=1;k<iq;k++)
                alf[k][iq]=pow(eps1,((a[k+1]-a[iq+1])/
                    ((a[iq+1]-a[1]+1.0)*(2*k+1))));
        }
        epsold=eps;
        nvold=nv;                               保存 nv.
        a[1] += nv;                             把雅可比计算量加到工作量系数中
        for (k=1;k<=KMAXX;k++) a[k+1]=a[k]+nseq[k+1];
        for (kopt=2;kopt<KMAXX;kopt++)
            if (a[kopt+1] > a[kopt]*alf[kopt-1][kopt]) break;
        kmax=kopt;
    }
    h=htry;
    for (i=1;i<=nv;i++) ysav[i]=y[i];
    jacobn(*xx,y,dfdx,dfdy,nv);                 计算雅可比
    if (*xx != xnew || h != (*hnext)) {
```

```
        first=1;
        kopt=kmax;
    }
    reduct=0;
    for (;;) {
        for (k=1;k<=kmax;k++) {
            xnew=(*xx)+h;
            if (xnew == (*xx)) nrerror("step size underflow in stifbs");
            simpr(ysav,dydx,dfdx,dfdy,nv,*xx,h,nseq[k],yseq,derivs);
            半隐式中点规则
            xest=SQR(h/nseq[k]);                程序其余部分和bsstep相同
            pzextr(k,xest,yseq,y,yerr,nv);
            if (k != 1) {
                errmax=TINY;
                for (i=1;i<=nv;i++) errmax=FMAX(errmax,fabs(yerr[i]/yscal[i]));
                errmax /= eps;
                km=k-1;
                err[km]=pow(errmax/SAFE1,1.0/(2*km+1));
            }
            if (k != 1 && (k >= kopt-1 || first)) {
                if (errmax < 1.0) {
                    exitflag=1;
                    break;
                }
                if (k == kmax || k == kopt+1) {
                    red=SAFE2/err[km];
                    break;
                }
                else if (k == kopt && alf[kopt-1][kopt] < err[km]) {
                        red=1.0/err[km];
                        break;
                    }
                else if (kopt == kmax && alf[km][kmax-1] < err[km]) {
                        red=alf[km][kmax-1]*SAFE2/err[km];
                        break;
                    }
                else if (alf[km][kopt] < err[km]) {
                    red=alf[km][kopt-1]/err[km];
                    break;
                }
            }
        }
        if (exitflag) break;
        red=FMIN(red,REDMIN);
        red=FMAX(red,REDMAX);
        h *= red;
        reduct=1;
    }
    *xx=xnew;
    *hdid=h;
    first=0;
    wrkmin=1.0e35;
    for (kk=1;kk<=km;kk++) {
        fact=FMAX(err[kk],SCALMX);
        work=fact*a[kk+1];
        if (work < wrkmin) {
            scale=fact;
            wrkmin=work;
            kopt=kk+1;
        }
    }
    *hnext=h/scale;
    if (kopt >= k && kopt != kmax && !reduct) {
        fact=FMAX(scale/alf[kopt-1][kopt],SCALMX);
        if (a[kopt+1]*fact <= wrkmin) {
            *hnext=h/fact;
            kopt++;
```

```
        }
    }
    free_vector(yerr,1,nv);
    free_vector(ysav,1,nv);
    free_vector(yseq,1,nv);
    free_vector(x,1,KMAXX);
    free_vector(err,1,KMAXX);
    free_matrix(dfdy,1,nv,1,nv);
    free_vector(dfdx,1,nv);
    free_matrix(d,1,KMAXX,1,KMAXX);
}
```

程序 **stifbs** 是一个解决刚性方程的极好的程序，和最好的 Gear 类型的程序可以相比。**stiff** 对于中等复杂度 $N$ 以及 $\epsilon \lesssim 10^{-4}$ 时还是很好的，当需要 $\epsilon \sim 10^{-8}$，**stifbs** 大概要快一个数量级。另外还有一些措施可用来提高 **stifbs** 的性能，例如，很偶然地，**simpr** 中的 **ludcmp** 会碰到奇异矩阵，这时可以让步长减小，可以减小当前 nseq[**k**]的一个因子。另外在某些问题上，还有步长稳定性的限制，至于怎样自动解决这些问题，参见[6]。

参考文献和进一步读物：


Gear, C. W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice Hall). [1]

Kaps, P., and Rentrop, P. 1979, *Numerische Mathematik*, vol. 33, pp. 56～68. [2]

Shampine, L. F. 1982, *ACM Transactions on Mathematical Software*, vol. 8, pp. 93～113. [3]

Enright, W. H., and Pryce, J. D. 1987, *ACM Transactions on Mathematical Software*, vol. 13 pp. 1～27. [4]

Bader, G., and Deuflhard, P. 1983, *Numerische Mathematik*, vol. 41, pp. 373～398. [5]

Deuflhard, P. 1987, "Uniqueness Theorems for Stiff ODE Initial Value Problems," *Preprint SC* 87～3 (Berlin: Konrad Zuse Zentrum fur Informationstechnik). [6]


## 16.7 多步法、多值法和预测校正法

多步法和多值法描述了两类不同的用于解常微分方程的基本方法，预测-校正法是这些方法的一种特殊情况——实际上也是最广泛出现的情况。相应地，预测-校正算法有时泛指所有这些方法。

我们怀疑预测-校正算法已经过时，而且对大多数常微分方程来说已不是最佳解法了。对需要高精度的情况，或是对方程右边算式的估值计算量很大的情况，布里斯奇-斯托算法占据主要地位。对于比较简单或低精度的问题，自适应步长的龙格-库塔法占据主要地位。我们觉得预测-校正算法是介于其间的一种算法，也许只有一种情况最适合于此：需要比较高的精度，函数很平滑而且等式右边很复杂。这种情况我们在下面讨论。

不管怎么说，这些方法都经历了很长的历史过程，现在很多教科书上都是这些内容，而且有很多标准的常微分方程都是基于预测-校正算法解决的。很多能干的研究者用预测-校正算法很经验，这些人没有必要马上改变原来的习惯。然而，作为一个知识广泛的实践者，最好也能熟悉一下这些方法涉及的原理，以及这些方法局限性的各个细节。不然的话，当第一次突然碰到需用预测-校正算法编程时，会感到束手无策的。

我们首先考虑多步法逼近。考虑一下求解一个常微分方程和求函数积分有什么不同：对于一个函数，被积函数和自变量的关系是已知的，而且是可估算的。对于一个常微分方程，

"被积函数"是等式的右端部分,既和自变量 $x$ 也和因变量 $y$ 有关系,因此要计算 $y'=f(x,y)$的从 $x_n$ 到 $x$ 的解.我们有

$$y(x)=y_n+\int_{x_n}^{x} f(x',y)dx' \qquad (16.7.1)$$

在象龙格-库塔或布里斯奇-斯托法的单步计算中,在 $x_{n-1}$处的 $y_{n+1}$值只依赖于 $y_n$ 的值。在多步法中,我们用通过前面几个点 $x_n,x_{n-1},\ldots$ 或也可能通过 $x_{n+1}$的一个多项式来近似 $f(x,y)$。计算 $x=x_{n-1}$处式(16.7.1)的积分,则成为下列形式

$$y_{n+1}=y_n+h(\beta_0 y'_{n+1}+\beta_1 y'_n+\beta_2 y'_{n-1}+\beta_3 y'_{n-2}+\cdots) \qquad (16.7.2)$$

这里 $y'_n$ 代表 $f(x_n,y_n)$依此类同。如果 $\beta_0=0$,该方法是显式的;否则是隐式的。该方法的阶次决定于我们用了多少已走的步数来得到 $y$ 的每个新值。

考虑一下,我们怎样才能解一个形为式(16.7.2)的隐式公式求出 $y_{n-1}$。有两种方法:**函数迭代法**和**牛顿法**。在函数迭代法中,我们取 $y_{n-1}$的某些初始猜测值,将其插入到式(16.7.2)的右端,以得到 $y_{n+1}$的变动值,再把该变动值代回右边,继续迭代。但我们如何得到 $y_{n+1}$的初始猜测值?很容易!只需用类似式(16.7.2)的显式即可,这叫做**预测步**。在预测步中,我们主要是外推多项式,以拟合从以前点到新点 $x_{n+1}$的导数值,并从 $x_n$ 到 $x_{n+1}$进行类似辛普森法求积分。随后的类似辛普森法的积分,用 $y_{n+1}$的预测步值来内插导数,被称为**校正步**。预测步和校正步的函数值之差提供了局部截断误差的信息,可用于误差控制及调整步长。

如果一个校正步成功了,多几个校正步会不会更好些?为何不用校正步作为一个改进的预测步并在每步迭代到收敛呢?答案是:即使有一个十全十美的预测,该步的精度仍只可能达到校正步的有限阶次的精度。其不能矫正的误差项和迭代能消除的误差是同阶的,所以最好只把误差项前的系数缩小为原来的几分之几。因此毫无疑问,进一步的提高是不值得的,最好还是把精力放在缩小步长上面。

迄今为止,读者也许想到值得而且必需在每一步之前预测几个间隔点,然后在类似辛卜生法的校正中;用不同的权重来使用所有这些间隔点。这并不是一个好主意。外推是这种方法中最不稳定的部分,应该尽量减少它的影响。因此,预测-校正算法中的积分步骤是重叠进行的,每一步都涉及几个步长间隔 h,每一步只比前一步扩展了一个这样的步长间隔,而且只有那个被扩展的间隔是通过每次预测步进行外推的。

最常用的预测-校正算法是用的所谓阿达姆斯-巴雪福斯-莫尔顿方法。这种方法有很好的稳定特性。阿达姆斯-巴雪福斯部分是预测步。例如,三阶的情况是:

$$\text{预测步:}\quad y_{n+1}=y_n+\frac{h}{12}(23y'_n-16y'_{n-1}+5y'_{n-2})+O(h^4) \qquad (16.7.3)$$

其中在当前点 $x_n$ 的信息和前两个点 $x_{n-1}$和 $x_{n-2}$(假设设置等间距)的信息,一起用来预测在下一点 $x_{n+1}$上的值 $y_{n-1}$。

阿达姆斯-莫尔顿部分是校正步。三阶的情况是:

$$\text{校正步:}\quad y_{n+1}=y_n+\frac{h}{12}(5y'_{n+1}+8y'_n-y'_{n-1})+O(h^4) \qquad (16.7.4)$$

如果等式右端没有加入从预测步获得的 $y_{n+1}$试用值的话,校正步将会变成一个蕴含 $y_{n+1}$的很难处理的式子。

这种算法实际上要进行三个单独的步骤:预测步,我们称为 P;从最新的 $y$ 值估算出导

数值 $y'_{n+1}$，我们称之为 E；以及校正步，我们称之为 C。在这种表示中，用校正步进行 m 次迭代(这种作法我们曾批驳过)记作 $P(EC)^m$。也可以选择用一个 C 步或一个 E 步，最后一步 E 是最关键的，所以这种方法通常建议用 PECE。

注意，具有固定迭代数目的 PC 算法是一种显式算法！当我们事先固定了迭代的数目以后，则最后的值 $y_{n-1}$ 可以写作一些已知变量的复杂的函数式。因此固定迭代数目的 PC 算法失去了稳式方法的强稳定性，只应用于非刚性问题。

对于刚性问题，如果我们想要避免用细小步长的话，则必须用隐式方法(并非所有的隐式方法对于刚性问题都是很好的，幸运的是有一些已知的方法如 Gear 公式是很有效的)。这样，我们解决隐式方程便似乎有两种选择：函数迭代到收敛，或是牛顿迭代法。但是，对刚性问题来说，无论我们的预测是如此接近，函数迭代只有在我们用极小的步长才可能收敛。因此，牛顿迭代法通常是多步法解刚性问题的基本方法。对于收敛，牛顿法对步长的大小不敏感，只要预测足够精确就行。

多步法，就我们所述而言，在实现时有两个严重问题：

- 因为公式需要等间距步长的结果，所以步长调整比校困难。
- 开始和停止当前问题时有些麻烦。在开始时，我们需要初始值加上前面几步来起动。停止时也有问题，因为等长的步距不可能刚好落在停止点上。

过去 PC 算法的实施中，用一些繁琐的方法来处理这些问题。例如，可以用龙格-库塔法来开始和停止。步长的变化需经相当复杂的记录信息进行某些内插过程。幸运的是这些缺陷可以由多值法来消除。

对于多值算法而言，用于积分器的最基本的数据是，在当前点 $x_n$ 解的泰劳级数展开式的头几项。其目标是求得解及其在下一个点 $x_{n+1}$ 的展开式系数。这和多步法明显不同。在多步法中数据是在 $x_n, x_{n-1}, \cdots$ 上求解的值。我们通过考虑一个四值算法来说明这种思路。这个四值算法中，基本的数据是

$$\mathbf{y}_n \equiv \begin{bmatrix} y_n \\ h y'_n \\ (h^2/2) y''_n \\ (h^3/6) y'''_n \end{bmatrix} \tag{16.7.5}$$

同样也按惯例用 $h = x_{n-1} - x_n$ 的幂次来测度导数项，如上式所示。注意，这里我们用向量符号 $\mathbf{y}$ 代表其解在某一点的头几项导数，而并不是求解一个有很多分量 $y$ 的方程组。

在式(16.7.5)的数据形式中，我们可以近似某点 $x$ 处的解 $y$：

$$y(x) = y_n + (x - x_n) y'_n + \frac{(x - x_n)^2}{2} y''_n + \frac{(x - x_n)^3}{6} y'''_n \tag{16.7.6}$$

在式(16.7.6)中，设 $x = x_{n+1}$，即可得到 $y_{n+1}$ 的估计值。同样可以得到 $y'_{n+1}$ 以及 $y''_{n+1}$ 和 $y'''_{n+1}$。称其结果的近似值为 $\tilde{\mathbf{y}}_{n+1}$，这里波浪号提示我们所作的只是一个对解及其导数的多项式外推。目前我们还未用到差分方程，但很容易证明

$$\tilde{\mathbf{y}}_{n+1} = \mathbf{B} \cdot \mathbf{y}_n \tag{16.7.7}$$

其中矩阵 $\mathbf{B}$ 是

$$\mathbf{B} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{16.7.8}$$

现在，我们写出以后我们要用到的实际估计值 $\mathbf{y}_{n+1}$，它等于 $\tilde{\mathbf{y}}_{n+1}$ 增加一个很校正量：

$$\mathbf{y}_{n+1} = \tilde{\mathbf{y}}_{n+1} + \alpha \mathbf{r} \tag{16.7.9}$$

这里 **r** 是一个固定的数字向量，同样 **B** 也是一固定矩阵。可通过满足下面微分方程来固定 $\alpha$：

$$y'_{n+1} = f(x_{n+1}, y_{n+1}) \tag{16.7.10}$$

式(16.7.9)的第二个方程是

$$h y'_{n+1} = h\tilde{y}'_{n+1} + \alpha r_2 \tag{16.7.11}$$

这将和式(16.7.10)一致而提供

$$r_2 = 1, \quad \alpha = hf(x_{n+1}, y_{n+1}) - h\tilde{y}'_{n+1} \tag{16.7.12}$$

值 $r_1$、$r_3$和 $r_4$是任意的，可由给定的四值法选择。不同的选择给出不同的阶次的算法(例如，通过 $h$ 的阶次，最后的表达式(16.7.9)实际上近似其解)，以及不同的稳定性。

有一个很有意思的结果，我们解释中没有明显说明，就是多值法和多步法完全是等价的。换言之，由多值法给定 **B** 和 **r** 导出的 $y_{n+1}$的值和多步法中用式(16.7.2)给出 $\beta$ 而导出的值是一样的。例如，证明阿达姆-巴雪福斯公式(16.7.3)和一个 $r_1=0$，$r_3=3/4$，$r_4=1/6$ 的四值法是等价的。该算法是显式的，因为 $r_1=0$。阿达姆斯-莫尔顿方法即式(16.7.4)相当于 $r_1=5/12$、$r_3=3/4$及 $r_4=1/6$ 的隐式四值算法。隐式多值算法和隐式多步算法的处理方法一样；或者通过预测校正逼近，用显式方法作预测步；或者是用牛顿迭代解决刚性问题。

为什么我们要不厌其烦地介绍一个新方法，而其结果却和你已知道的方法等效呢！原因是多值法很容易解决上面提到的实现多步法时的两个困难。

首先考虑第一个问题步长调整。要在某一点 $x_n$ 把步长从 $h$ 变到 $h'$，只需要简单地把式(16.7.5)中 $\mathbf{y}_n$ 的元素乘上$(h'/h)$的适当的幂，便可连续到 $x_n+h'$ 了。

多值法也允许在算法阶次上进行相对简单的变化：只需简单地改变 **r**。这样做通常的办法是首先从误差估计中用现有的阶次决定新的步长，然后检查是应该用高一阶或低一阶的步长来预测。选择允许下一步取最大的那个阶次。能够选择阶次这就让起点问题也变得容易了、只需简单地从一阶算法开始，再让阶次自动增加到合适的阶次。

对于低精度的要求，类似 **rkqs** 的龙格-库塔程序几乎总是最有效的。对于高精度，**bsstep** 性能既好且有效。对于非常平滑的函数，一个变阶次的 PC 算法可达很高阶。如果等式右端相对较复杂，这样估值的计算量超过了记录信息的费用，则最好的 PC 程序在处理这类问题上比布里斯奇-斯托法要好。正如所想象的那样，这样变步长，高阶次的算法在程序中并不麻烦。如果用户觉得自己的问题适合于这种方法，我们建议采用 PC 软件包。详细情况参见 Gear[1]或 Shampine 和 Gordon[2]。

我们预计，象布里斯奇-斯托那样的外推算法是越来越精致而复杂，最终在所有应用方面都将丢弃 PC 算法，我们希望我们这点是对的。

参考文献和进一步读物：

Gear, C. W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 9. [1]

Shampine, L. F., and Gordon, M. K. 1975, *Computer Solution of Ordinary Differential Equations. The Initial Value Problem*, (San Francisco: W. H. Freeman). [2]

# 第十七章　两点边界值问题

## 17.0　引言

当常微分方程要求在自变量的两个或两个以上值上满足边界条件时，这种问题叫作**两点边界值问题**。正如该术语所表明的，目前最普遍的情况是要求在两个点——通常是积分的起点和终点上满足边界条件。但是"两点边界值问题"这一说法也广泛用于更加复杂的情况，如有的条件是在末端指定的，有的条件是在内部指定(通常是奇异点)。

初始值问题(第十六章)和两点边界值问题(本章)的主要区别在于，前者我们可以从其起点处(初始值)一个可接受的解开始，只需要通过数值积分到其终点(结果值)即可；而在现在情况下，起始点的边界条件并不是唯一决定求解开始的条件　而且在满足起始边界的解(不完全)中"随便"选择一个解，它几乎并不能满足其它特定点的边界条件。

一般情况下，需要用迭代法把这些空间上分散的边界值条件融合成一个单一的微分方程的解。这一点并不感到奇怪，正因为这个原因，两点边界值问题比初始值问题要更费精力。这必须在有意义的区间上对微分方程积分，或是执行类似"松驰"过程(参见下面)几次，有时甚至是很多次。往往只有在特定的线性微分方程情况下，才能事先说出需要多少次的迭代计算。

"标准"的两点边界值问题有如下的形式：我们需要求一组 $N$ 个联立的一阶常数微分方程的解，在起始点 $x_1$满足 $n_1$个边界条件，在最终点 $x_2$满足 $n_2=N-n_1$个边界条件(记住，所有高于一阶以上的微分方程都可以写成一组联立的一阶方程，参第16.0节)。

微分方程是

$$\frac{dy_i(x)}{dx}=g_i(x,y_1,y_2,\ldots,y_N)\qquad i=1,2,\ldots,N \tag{17.0.1}$$

在点 $x_1$，解需满足

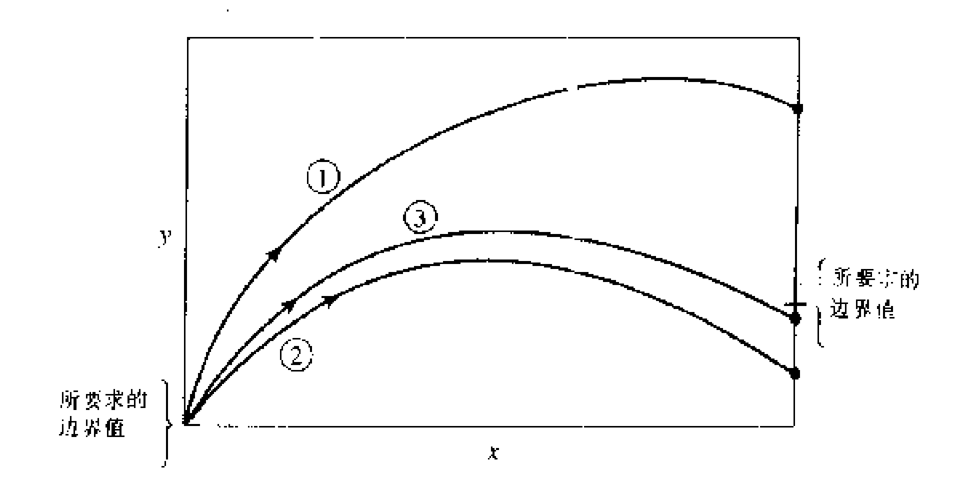
$$B_{1j}(x_1,y_1,y_2,\ldots,y_N)=0\qquad j=1,\ldots,n_1 \tag{17.0.2}$$

而在 $x_2$点，应满足

$$B_{2k}(x_2,y_1,y_2,\ldots,y_N)=0\qquad k=1,\ldots,n_2 \tag{17.0.3}$$

解决两点边界值问题有两种不同类型的数值解法，在**打靶法**(第17.1节)中，我们在一个边界上选择了所有应变量的值，这些值必须和该边界的边界条件保持一致；但是在另外一个边界上将应变量设置成依赖于任意参数，其参数值是最初"随机"猜测的。然后，我们用初始值的方法积分常微分方程，一直到达另外一个边界(或是定义了边界条件的内点)。一般说来，我们发现和那里的边界值不吻合，但我们遇到过多维寻解问题，就象在第7.6节和第9.7节中那样：找到起始点处的调整量，从而把在其它边界上的误差量减小到零。如果我们把微分方程的积分过程比喻为弹道轨迹的话，选择初始条件就相当于瞄准的过程(参见图17.0.1)。打靶法提供了有规律可循的逼近法，可以进行限定一定范围的"射击"，这可以有规律地

着进我们的"瞄准"作用。




在满足边界条件的某一端点试探性地从那一点开始积分，在另外一端点和所需边界条件的门广差用以调节起始条件，直到在两个端点的边界条件都被满足。

图17.0.1　打靶法(示意图)

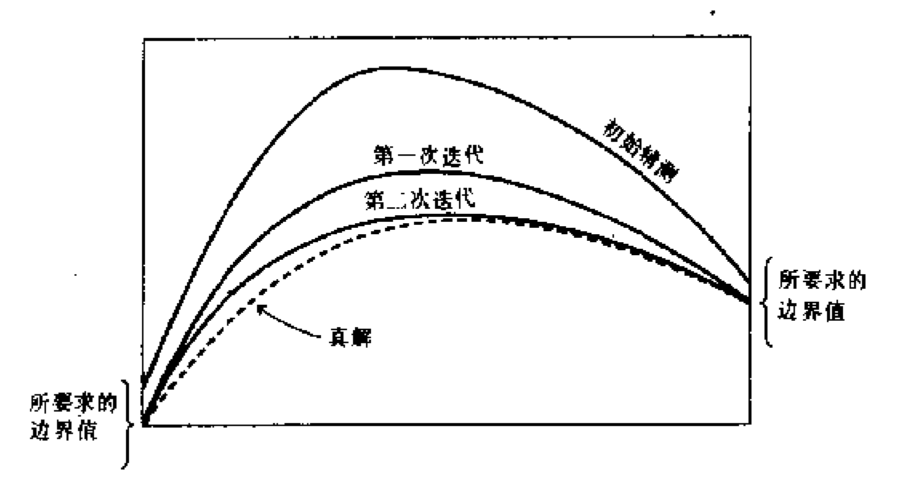
作为打靶法的另一变异(第17.2节)，我们可以猜测在定义域的两端都不知的自由参量的值，把方程积分到某一合适的中点，设法调整猜测的参数值让解在那一点上"平滑地"相连。在所有的打靶法算法中，试验解能够"精确地"(或考虑到积分算法是相对来说很精确地)满足微分方程，但是只有在迭代结束以后，试验解才能满足所需的边界值条件。

**松弛法**用了另外一种逼近方法。微分方程由覆盖积分限的一系列的有限个差分方程来代替。试验解由各个网格点上应变量的值组成，并不满足所需的有限个差分方程，甚至也不需要满足所需的边界条件。迭代过程，这里称作**松弛**过程，是调整所有在网格上的值，使他们满足各个联立的差分方程，同时也满足边界条件(参见图17.0.2)。例如，如果问题涉及三个联立的方程以及一百个网格点，则我们必须根据解先预测三百个变量的值，然后进行改进来得到最后的结果。

需要这么多调整，你也许会奇怪，松驰法是有效的方法吗?但(对某些问题来说)它的确是一种很有效的方法!当边界值条件特别棘手而错综复杂的时候，或是涉及到不能用封闭形式进行求解的代数关系时，松驰法比打靶法的效果要好。如果解是平滑的而且不是高度振荡的情况，则松驰法效果最好。如果振荡得很厉害就需要很多的网格点来进行精确的表示。所需要的网格点的数目和位置也许都是事先未知的。这种情况下用打靶法通常比较好，因为打靶法中的变步长积分能够和解的特点相适应。

人们通常喜欢将松驰法用于常微分方程组的解有另外一部分，这部分不出现在满足所有边界条件的最后结果中，但在用打靶法进行初始值积分时它却会造成很多问题。典型的情况，是一个衰减的指数解中会有一个增长的指数部分。

为使松驰法有效地发挥作用，良好的初始预测值是一个关键。通常情况下，人们常常需要多次解决同一个问题，每次都是某几个参数略微变一下。在这种情况下，当参数改变时，原先的结果也许就是一个好的初始预测值，并且松驰法能够有效地解决问题。




预测一初始值近似满足微分方程和边界条件，用迭代过程对其进行调整直到最后和真实完全吻合。

图17.0.2　松驰法(示意图)

在挑选这两种方法时，如果还不太有经验，那么可以采用我们的建议：首先选用打靶法，然后再用松驰法。

### 17.0.1　能用标准边界值问题求解的问题

有两类重要的问题能够用方程(17.0.1)～(17.0.3)所描述的标准边界值问题来求解。第二类是微分方程组的特征值问题，这里的微分方程组的右端依赖于参数 $\lambda$

$$\frac{dy_i(x)}{dx} = g_i(x, y_1, \ldots, y_N, \lambda) \tag{17.0.4}$$

而且必须满足 $N+1$个边界条件，而不只是 $N$ 个。这种问题是超定性的，通常不是对任意 $\lambda$ 值都有解，对某些特殊的 $\lambda$ 值，即特征值，方程式(17.0.4)确实有解。

我们引入一个新的应变量和另一个微分方程来推导这个问题

$$y_{N+1} \equiv \lambda \tag{17.0.5}$$

$$\frac{dy_{N+1}}{dx} = 0 \tag{17.0.6}$$

在第17.4节中给出了这种处理方法的一个例子。

另外一种能化成标准形式的是**自由边界值问题**。这里边界只有一个边界横坐标 $x_1$是指定的，而另一个 $x_2$是未确定的。这样使式(17.0.1)具有一个满足总共是 $N+1$个边界条件的解。这里我们再次加入一个额外的定值应变量：

$$y_{N+1} \equiv x_2 - x_1 \tag{17.0.7}$$

$$\frac{dy_{N+1}}{dx} = 0 \tag{17.0.8}$$

我们还定义了一个新自变量 $t$，并设置成

$$x - x_1 \equiv t\, y_{N+1}, \qquad 0 \le t \le 1 \tag{17.0.9}$$

原来总共 $N+1$个关于 $dy_i/dt$ 的微分方程，现在即成标准形式，其中 $t$ 在已知区间0到1之间

变化。

## 17.1 打靶法

在这一节中，我们讨论"纯正"的打靶法，其中积分过程是从 $x_1$ 到 $x_2$，并且我们努力使积分结果在积分的终点和边界条件匹配。在下一节里，我们介绍射向某一中间合适点的情况，从区间的两端"射击"，争取在某一中间点满足连续条件。这样才求出满足方程和边界值条件的解。

我们实现的打靶法精确地运用了多维全局收敛牛顿-拉斐森法（第9.7节），它设法零化 $n_2$ 个变元的 $n_2$ 个函数，这些函数从 $x_1$ 到 $x_2$ 积分 $N$ 个微分方程得到。我们看一下其原理。

在初始点 $x_1$ 有 $N$ 个起始值 $y_i$ 要进行定义，但必须满足 $n_1$ 个条件，因此这就有 $n_2=N-n_1$ 个自由定义的起始值。我们假设这些自由定义的值是 $n_2$ 维向量空间的一个向量 **V**，当知道了边界条件的函数形式(17.0.2)，即可作出一函数，该函数生成 $N$ 个起始值 **y** 的完备集，并在 $x_1$ 点满足边界条件。该函数可以从任一向量 **V** 来生成。在该向量，对其中的 $n_2$ 个元素没有任何限制。换言之，式(17.0.2)转换成一个表达式

$$y_i(x_1) = y_i(x_1; V_1, \dots, V_{n_2}) \qquad i = 1, \dots, N \tag{17.1.1}$$

下面实现式(17.1.1)的函数叫作 **load**。

注意 **V** 的元素也许刚好是 **y** 的某些"自由"分量值，**y** 的其它分量则由边界条件来确定。另一方面，**V** 的元素也许可以以某种简便的方式，将构成满足起始边界条件的解参数化。边界值条件经常用一些代数关系来约束 $y_i$，而不是单个地规定它们的特定值。对某一组 $y_i$，用一些附加的参数集往往比较容易"解决"边界值关系。无论采用哪种方法都可以，只要能通过向量空间 **V**（用式(17.1.1)）生成所有允许的起始向量 **y**。

给定一个特定的 **V**，特定的 $\mathbf{y}(x_1)$ 即生成了。然后按初始值问题对常微分方程积分到 $x_2$（例如可用第十六章的 **odeint**），就可以将它转变成了 $\mathbf{y}(x_2)$。在 $x_2$ 点，我们定义一个**偏差向量 F**，也是 $n_2$ 维的，其分量用来测度离在 $x_2$ 点处满足 $n_2$ 个边界条件(17.0.3)的差距。最简单的方法是用式(17.0.3)的右端，

$$F_k = B_{2k}(x_2, \mathbf{y}) \qquad k = 1, \dots, n_2 \tag{17.1.2}$$

对于向量空间 **V**，也可以使用任何其它的参数化方法，只要 **F** 空间能够扩张成所要求边界条件可能的偏差空间，而且当且仅当在 $x_2$ 点的边界条件得到满足时，**F** 的所有分量等于零。下面，将要求提供一个用户自编的函数 **score**，用式(17.0.3)把终止值 $\mathbf{y}(x_2)$ 的 $N$ 维向量转换成一个 $n_2$ 维偏差向量 **F**。

现在，就牛顿-拉斐森法而言，我们就得费一些周折了，我们需要找到一个 **V** 的向量值，让其使 **F** 向量值趋于零。我们通过调用在第9.7节中，程序 **newt** 实现的全程收敛牛顿法来完成这一点。牛顿法的核心涉及到求解 $n_2$ 个线性方程组

$$\mathbf{J} \cdot \delta\mathbf{V} = -\mathbf{F} \tag{17.1.3}$$

的解，然后返回加上校正量

$$\mathbf{V}^{new} = \mathbf{V}^{old} + \delta\mathbf{V} \tag{17.1.4}$$

在式(17.1.3)中，雅可比矩阵 **J** 的元素由下式决定：

$$J_{ij} = \frac{\partial F_i}{\partial V_j} \tag{17.1.5}$$

用解析法计算这些偏导数是不现实的，而且每个值都需要对 N 个常微分方程的单独积分，而且接着估值

$$\frac{\partial F_i}{\partial V_j} \approx \frac{F_i(V_1,\ldots,V_j+\Delta V_j,\ldots)-F_i(V_1,\ldots,V_j,\ldots)}{\Delta V_j} \tag{17.1.6}$$

这已在 **newt** 程序下的 **fdjac** 程序中自动完成。用户需要提供 **newt** 的输入只是程序 **vecfunc**，它是通过常微分方程组求积来计算 **F**。下面有一个合适的程序，称为 **shoot**，它作为一个实际参量传递到程序 **newt** 中。

```
#include "nrutil.h"
#define EPS 1.0e-6

extern int nvar;                  用户需在主程序中定义和设置的变量
extern float x1, x2;

int kmax, kount                   与程序 odeint 传递
float *xp, **yp, dxsav;

void shoot (int n, float v[], float f[])
```

对 nvar 个联立常微分方程用 newt 程序通过从 x1到 x2打靶解决两点边界值问题的程序。在 x1处 nvar 个常微分方程的初始值是采用用户自编程序 **load**，利用 n2个输入系数 v[1..n2]来产生。程序积分常微分方程采用步格-库塔法，容差限为 eps，初始步长 h1，最小步长 hmin。在 x2点调用用户自编程序 **score** 来计算 n2个函数 f[1..n2]。f[1..n2]应该在满足 x2处边界条件时变为零。函数 f 在输出返回。newt 用全程收敛牛顿法来调整参数 v 直到 f 为零。用户提供的程序 **derivs**(x,y,dydx)提供常微分方程积分求导数信息(见第十六章)。上面第一组全程变量从主程序中接收值，以便 **shoot** 中能有 **newt** 的参数 vecfunc 所需的语法。

```
{
    void derivs(float x, float y[], float dydx[]);
    void load(float x1, float v[], float y[]);
    void odeint(float ystart[], int nvar, float x1, float x2,
        float eps, float h1, float hmin, int *nok, int *nbad,
        void (*derivs)(float, float [], float []),
        void (*rkqs)(float [], float [], int, float *, float, float,
        float [], float *, float *, void (*)(float, float [], float [])))
    void rkqs(float y[], float dydx[], int n, float *x,
        float htry, float eps, float yscal[], float *hdid, float *hnext,
        void (*derivs)(float, float [], float []));
    void score(float xf, float y[], float f[]);
    int nbad,nok;
    float h1,hmin=0.0,*y;

    y=vector(1,nvar);
    kmax=0;
    h1=(x2-x1)/100.0;
    load(x1,v,y);
    odeint(y,nvar,x1,x2,EPS,h1,hmin,&nok,&nbad,derivs,rkqs);
    score(x2,y,f);
    free_vector(y,1,nvar);
}
```

对有些问题来说，初始化步长值也许与初始条件有很密切的关系。最直接的作法是修改 **load** 函数，使其包括一个作为另一个输出变量的参考步长 h1，并通过全程变量把它反馈给 **fdjac**。

这样一个完整的打靶法循环过程需要对 $N$ 个联立常微分方程进行 $n_2+1$ 次积分：一次积分用作估计当前误差程度，$n_2$ 次对偏导数积分。每一次新的循环过程要求新的一轮 $n_2+1$ 次积分。这也说明了两点边界值问题和初始值问题比较起来的确需要更大量额外的工作量。

如果微分方程线性的，则只需要一个完整的循环过程，因为式(17.1.3)～(17.1.4)可以直接让我们能取得正确结果。但是再进行一次循环也是有用的，这可以减少部分(但不可能是全部)的舍入误差。

作为用户，必须为 **shoot** 程序提供：(i)函数 **load**(x1,v,y)，用以计算 n 维向量 y[1..n](当然满足起始边界条件)，在初始点 x1给出 v[1..n2]的自定义变量；(ii)函数 **score**(x2,y,f)，用以计算在终端边界条件偏差向量 f[1..n2]，在终端点 x2给出向量 y[1..n]；(iii)起始向量 v[1..n2]；(iv)函数 **derivs** 进行常微分方程的积分；以及其它在上面程序开头所说明的一些参数。

在第17.4节中，我们将列举一个简单的程序，用以说明怎样使用程序 **shoot**。

## 17.2 对合适点打靶

在第17.1节中，介绍的打靶法约定了"射击"轨迹能够穿越整个的积分域，很快即可以收敛到正确的解上。在有些问题发生的情况是，因为错误严重的起始条件，初始解从 $x_1$ 到 $x_2$ 要碰到某些不可计算的，或是灾难性的结果。例如，某个平方根变成负数，导致了数值代码的混乱，使简单的打靶法将陷入困境。

一种不同的但是相关的情况是，两个端点都是常微分方程组的奇异解。通常需要用特别的方法在奇异点附近积分，例如用解析渐近法。在这样的情况下，沿着背离奇异点方向进行积分是可行的，用某些特殊的方法经过第一个点旁边一点点，然后求出"初始值"进行进一步的数值积分。但是，通常积分到某一奇异点是不行的，因为用户通常不用费很大的精力就得到奇异点附近的"错误"解的解析展开式(这些解并不满足所需的边界条件)。

解决上面所提困难的方法是用对**合适点打靶**的方法。我们并不从 $x_1$ 积分到 $x_2$，而是首先从 $x_1$ 积分到 $x_1$ 和 $x_2$ 之间的一点 $x_f$；然后再从 $x_2$(沿反方向)积到 $x_f$。

如果和以前一样，在 $x_1$ 点的边界条件数是 $n_1$，则在起始 $x_1$ 有 $n_2$ 个可自由设定的初始值，而在 $x_2$ 有 $n_1$ 个可自由设定的起始值。(如果这里你感到有些疑惑，可以回到第17.1节。)我们因此可以在 $x_1$ 处定义一个 $n_2$ 维起始参数向量 $\mathbf{V}_{(1)}$，以及一个把 $\mathbf{V}_{(1)}$ 映射到在 $x_1$ 处满足边界条件的向量 **y** 的方法 **load1**(x1,v1,y)。

$$y_i(x_1) = y_i(x_1; V_{(1)1}, \ldots, V_{(1)n_2}) \qquad i = 1, \ldots, N \tag{17.2.1}$$

同样，我们在 $x_2$ 处也能定义一个 $n_1$ 维起始参数向量 $\mathbf{V}_{(2)}$，以及一个把 $\mathbf{V}_{(2)}$ 映射到在 $x_2$ 处满足边界条件的向量 **y** 的方法 **load2**(x2,v2,y)

$$y_i(x_2) = y_i(x_2; V_{(2)1}, \ldots, V_{(2)n_1}) \qquad i = 1, \ldots, N \tag{17.2.2}$$

这样，把 $\mathbf{V}_{(1)}$ 和 $\mathbf{V}_{(2)}$ 结合起来，我们就总共有 $N$ 个可调整的参数。必须要满足的 $N$ 个条件是，从两头积分得到的，在 $x_f$ 点处 **y** 向量的 $N$ 个分量应该相同，即

$$y_i(x_f; \mathbf{V}_{(1)}) = y_i(x_f; \mathbf{V}_{(2)}) \qquad i = 1, \ldots, N \tag{17.2.3}$$

在有些问题中，这 $N$ 个匹配条件最好用 $N$ 个不同函数 $F_i(i=1\cdots N)$(物理地、数学地或数值

地)进行描述,每个函数也许都和 $y_i$ 的 $N$ 个元素有关。在这些情况下,式(17.2.3)由下式替代:

$$F_i[\mathbf{y}(x_f;\mathbf{V}_{(1)})] = F_i[\mathbf{y}(x_f;\mathbf{V}_{(2)})] \quad i = 1,\ldots,N \tag{17.2.4}$$

在下面的程序中,用户提供的函数 **score**(xf,y,f)是用来把输入的 $N$ 维向量 **y** 映射成为输出的 $N$ 维向量 **F**。大多数情况下,可以把这个函数直接进行相等映射。

射向某一合适点的方法和第17.1节中所使用的牛顿-拉斐森方法完全一样,参考前一节的程序 **shoot**,理解下面的程序 **shootf** 应该没有什么困难。在使用时的主要区别是,用户必须提供 **load1** 以及 **load2**。而且,在调用程序中必须给 v1[1..n2]和 v2[1..n1]提供初始猜测值,在第17.4节中也给出了一示例程序说明如何射向一合适点。

```
#include "nrutil.h"
#define EPS 1.0e-6

extern int nn2,nvar;                用户需要在主程序中定义和设置的变量
extern float x1,x2,xf;

int kmax, kount;                    与程序 odeint 传递
float *xp, **yp,dxsav

void shootf(int n, float v[], float f[])
    用 newt 求解 nvar 个联立常微分方程组两点边界值问,从 x1和 x2射向某一合适点 xf 的程序。nvar 个常微分方程组
    x1(x2)的初始值由 n2(n1)个系数 v1(v2)产生,它使用用户提供的程序 load1(load2)获得。在主程序中,通过语句 v1
    =v 和 v2=&v[n2]把系数 v1和 v2存储在单个矩阵 v[1..n1+n2]中。输入参数 n=n1+n2=nvar。程序使用龙格-
    库塔法积分常微方程到 xf,其容差限为 eps,初始步长 h1及最小步长 hmin。在 xf 处调用用户定义的程序 score,以计
    算 nvar 个函数 f1和 f2,这些函数应在 xf 匹配。f 的差值应返回到输出。newt 采用全局收敛牛顿法,以调整 v 值,直到
    函数 f 等于零。用户提供的程序 derivs(x,y,dydx)给出常微分方程的导数信息(参见第十六章)。上面的第一个全局
    变量是从主程序中接收它的值,以使 shoot 能有 newt 的参数 vecfunc 的所要求的语法。在主程序中设 nn2=n1;
{
    void derivs(float x, float y[], float dydx[]);
    void load1(float x1, float v1[], float y[]);
    void load2(float x2, float v2[], float y[]);
    void odeint(float ystart[], int nvar, float x1, float x2,
        float eps, float h1, float hmin, int *nok, int *nbad,
        void (*derivs)(float, float [], float []),
        void (*rkqs)(float [], float [], int, float *, float, float,
        float [], float *, float *, void (*)(float, float [], float [])));
    void rkqs(float y[], float dydx[], int n, float *x,
        float htry, float eps, float yscal[], float *hdid, float *hnext,
        void (*derivs)(float, float [], float []));
    void score(float xf, float y[], float f[]);
    int i,nbad,nok;
    float h1,hmin=0.0,*f1,*f2,*y;

    f1=vector(1,nvar);
    f2=vector(1,nvar);
    y=vector(1,nvar);
    kmax=0;
    h1=(x2-x1)/100.0;
    load1(x1,v,y);                  用最佳试验值v1从x1到xf的路径
    odeint(y,nvar,x1,xf,EPS,h1,hmin,&nok,&nbad,derivs,rkqs);
    score(xf,y,f1);
    load2(x2,&v[nn2],y);            用最佳试验值v2从x2到xf的路径
    odeint(y,nvar,x2,xf,EPS,h1,hmin,&nok,&nbad,derivs,rkqs);
    score(xf,y,f2);
```

```
    for (i=1;i<=n;i++) f[i]=f1[i]-f2[i];
    free_vector(y,1,nvar);
    free_vector(f2,1,nvar);
    free_vector(f1,1,nvar);
}
```

有的边界值问题，即使采用射向某一合适点也会出现问题——积分区间不得不用几个合适点进行分段，而解则必须在每个合适点上得到匹配，详见[1]。

参考文献和进一步读物：


Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §§7.3.5～7.3.6[1]


## 17.3 松驰法

在松驰法中，我们把常微分方程用某些栅格或网格点上的近似有限差分方程组(FDEs)来代替，这些网格点张成了这个积分域。作为一个范例，我们可以把一个一般的一阶微分方程

$$\frac{dy}{dx} = g(x,y) \tag{17.3.1}$$

用一个与在两点 $k$，$k-1$ 函数值有关的代数方程

$$y_k - y_{k-1} - (x_k - x_{k-1})g\left[\frac{1}{2}(x_k + x_{k-1}),\frac{1}{2}(y_k + y_{k-1})\right] = 0 \tag{17.3.2}$$

来代替。在式(17.3.2)中有限差分方程的形式只说明这种思想，但其形式并非唯一的：有很多方法可以把常微分方程转换成有限差分方程。当遇到用栅格上的 $M$ 个点代替 $N$ 个联立的一阶常微分方程的问题时，其解是 $M$ 个栅格点上的 $N$ 个应变函数的值，也就是总共 $N\times M$ 个变量。松驰法和我们以前介绍过的方法一样，首先预测一个解，然后对其进行迭代和修正以达到真实解。当用迭代法修正解时，结果便逐渐逼近到真实解。

我们可以用很多种迭代法，但对大多数问题来说，我们古老的、标准的多维牛顿法效果很好，这种方法需要求解一个矩阵方程，但该矩阵是一种特殊的"块对角化"形式，在求解的时候，可能比一般的 $(MN)\times(MN)$ 的矩阵在时间和存储空间上都要节省很多，因为 $MN$ 很容易就达几千，这对于方法是否可行关系很密切。

在大多数成对的点上，我们的实施过程都是匹配的，就象在方程(17.3.2)中一样。匹配的点越多，则更增加了方法的复杂性。如果读者有耐心去编程，我们可以提供足够的知识背景。

我们来求解一组由有限差分方程替代常微分方程的一般的代数方程。常微分方程问题和式(17.0.1)～(17.0.3)所表示的完全一样。这里，我们有 $N$ 个联立的一阶方程，在 $x_1$ 点满足 $n_1$ 个边界条件，在 $x_2$ 点满足 $n_2=N-n_1$ 个边界条件。首先我们定义了一组 $k=1,2,\cdots,M$ 个点的栅格，我们提供了在这栅格组点上相应的变量 $x_k$ 的值。特别地，$x_1$ 是初始边界值，$x_m$ 是终点边界值。我们用 $\mathbf{y}_k$ 表示在点 $x_k$ 处应变量 $y_1$，$y_2$，…，$y_N$ 整组的值。在网格中间的任一点 $k$，我们可以用下列代数关系式来逼近 $N$ 个一阶常微分方程的方程组：

$$0 = \mathbf{E}_k \equiv \mathbf{y}_k - \mathbf{y}_{k-1} - (x_k - x_{k-1})\mathbf{g}_k(x_k,x_{k-1},\mathbf{y}_k,\mathbf{y}_{k-1}),\quad k = 2,3,\cdots,M \tag{17.3.3}$$

这个表达式表明 $\mathbf{g}_k$ 能用两点 $k$，$k-1$ 的信息进行计算，由 $\mathbf{E}_k$ 表示的有限差分方程提供了点 $k$，$k-1$ 的包含 $2N$ 个变量的 $N$ 个方程。(17.3.3)形式的差分方程能解出 $M-1$ 个点，$k=2,3,\cdots,M$。这样，有限差分方程提供了 $(M-1)N$ 个方程，求解 $MN$ 个未知变量，其余的 $N$ 个方程从边界条件导出。

在第一个边界值条件中，我们有

$$0 = \mathbf{E}_1 \equiv \mathbf{B}(x_1,\mathbf{y}_1) \tag{17.3.4}$$

在第二个边界值条件中，有

$$0 = \mathbf{E}_{M+1} = \mathbf{C}(x_M, \mathbf{y}_M) \qquad (17.3.5)$$

向量 E 和 B 只有 $n_1$ 个非零元，这对应于在 $x_1$ 点的 $n_1$ 个边界条件。把这些非零元放在最后的 $n_1$ 个元素中以后会有用的。也就是说，只对 $j=n_2+1, n_2+2, \cdots, N$ 来说 $E_{j,1} \neq 0$。在另一个边界条件中，只有 $\mathbf{E}_{M+1}$ 和 C 的最初 $n_2$ 个单元是非零的：$E_{j,M+1}$ 只对 $j=1,2,\cdots n_2$ 不为零。

在式(17.3.3)～(17.3.5)中的有限差分方程问题的"解"由一组 $y_{j,k}$ 变量的值组成，也就是由在 $M$ 个点 $x_k$ 上 $N$ 个变量的值组成。这样我们即可算出增量 $\Delta y_{j,k}$，而 $y_{j,k}+\Delta y_{j,k}$ 便是一个改进的近似解。

增量方程可以通过用相对较小的变化量 $\Delta \mathbf{y}_k$ 展开的一阶泰劳展开式来求解。在某一内点，$k=2,3,\cdots,M$，为：

$$\mathbf{E}_k(\mathbf{y}_k+\Delta\mathbf{y}_k, \mathbf{y}_{k-1}+\Delta\mathbf{y}_{k-1}) \approx \mathbf{E}_k(\mathbf{y}_k, \mathbf{y}_{k-1}) + \sum_{n=1}^{N} \frac{\partial \mathbf{E}_k}{\partial y_{n,k-1}} \Delta y_{n,k-1} + \sum_{n=1}^{N} \frac{\partial \mathbf{E}_k}{\partial y_{n,k}} \Delta y_{n,k} \qquad (17.3.6)$$

对真实解来说，我们希望修正值 $\mathbf{E}(\mathbf{y}+\Delta\mathbf{y})$ 变成零。这样，一般的方程组在某一内点能写成如下矩阵形式：

$$\sum_{n=1}^{N} S_{j,n} \Delta y_{n,k-1} + \sum_{n=N+1}^{2N} S_{j,n} \Delta y_{n-N,k} = -E_{j,k}, \quad j=1,2,\ldots,N \qquad (17.3.7)$$

这里：

$$S_{j,n} = \frac{\partial E_{j,k}}{\partial y_{n,k-1}}, \quad S_{j,n+N} = \frac{\partial E_{j,k}}{\partial y_{n,k}}, \quad n=1,2,\ldots,N \qquad (17.3.8)$$

在每个点 $k$ 的 $S_{j,n}$ 参量为一个 $N \times 2N$ 的矩阵。这样，每个内点提供一组 $N$ 个方程，它把 $2N$ 个校正量匹配到点 $k, k-1$ 的解的变量上。

同样，边界值的代数关系式可以展开成一阶泰劳展开式来修正解。因为 $\mathbf{E}_1$ 只依赖于 $\mathbf{y}_1$ 而变。在第一个边界条件，我们发现有：

$$\sum_{n=1}^{N} S_{j,n} \Delta y_{n,1} = -E_{j,1}, \quad j=n_2+1, n_2+2, \ldots, N \qquad (17.3.9)$$

其中

$$S_{j,n} = \frac{\partial E_{j,1}}{\partial y_{n,1}}, \qquad n=1,2,\ldots,N \qquad (17.3.10)$$

在二个边界条件处，有：

$$\sum_{n=1}^{N} S_{j,n} \Delta y_{n,M} = -E_{j,M+1}, \quad j=1,2,\ldots,n_2 \qquad (17.3.11)$$

其中

$$S_{j,n} = \frac{\partial E_{j,M+1}}{\partial y_{n,M}}, \qquad n=1,2,\ldots,N \qquad (17.3.12)$$

这样，我们通过等式(17.3.7)～(17.3.12)得到一组线性方程，可求解校正量 $\Delta\mathbf{y}$，迭代直到校正量足够小为止。这个方程组的结构比较特殊，因为每个 $S_{j,n}$ 只和点 $k, k-1$ 有关。图17.3.1说明了五个变量和四个网格点，且在第一边界处满足三个边界条件，在第二边界处满足两个边界条件的完全矩阵方程的典型情况。在矩阵左上角的那一块 $3\times5$ 的项来自点 $k=1$ 的边界条件 $S_{j,n}$；另外三个 $5\times10$ 的块是在内部点的 $S_{j,n}$，而且它在网点(2,1)、(3,2)和(4,3)上把变量联立组合起来；最后一块是满足第二边界值条件的项。

我们利用高斯消元法来求解方程(17.3.7)～(17.3.12)，求出增量 $\Delta\mathbf{y}$。这种方法利用了矩阵的特殊结构把总的计算量减至最少。而且这种方法通过把元素合并成一个特殊的块结构，而把矩阵系数的的存储量减至最小(如果对高斯消元法不太熟悉，可以参见第二章尤其是第2.2节)。高斯消元法求解方程组时，用了初等运算的操作，如通过其行同除以某一公因子，使在对角线上产生单位元，并再加上其它行的适当倍数，把对角线以下的元素化为零。这里，我们利用了块结构的优点，比纯粹高斯消元法进一步简化，这样系数存

储量也减少了。图17.3.2表示我们通过消元想要得到的形式，这是在回代步骤以前的形式。在消元过程中，只有很小的一个经约简后的 MN×MN 矩阵的元素需要进行存储。一旦矩阵变成了图17.3.2的形式，通过回代过程很快即可以得到解。

```
X X X X X                                      V    B
X X X X X                                      V    B
X X X X X                                      V    B
X X X X X X X X X X                            V    B
X X X X X X X X X X                            V    B
X X X X X X X X X X                            V    B
X X X X X X X X X X                            V    B
X X X X X X X X X X                            V    B
          X X X X X X X X X X                  V    B
          X X X X X X X X X X                  V    B
          X X X X X X X X X X                  V    B
          X X X X X X X X X X                  V    B
          X X X X X X X X X X                  V    B
                    X X X X X X X X X X        V    B
                    X X X X X X X X X X        V    B
                    X X X X X X X X X X        V    B
                    X X X X X X X X X X        V    B
                    X X X X X X X X X X        V    B
                              X X X X X        V    B
                              X X X X X        V    B
```

x 代表有限差分方程组的系数，v 代表解向量中的未知元，B 表示已知的方程右端的值，空白区表示为零。该矩阵方程将通过特殊形式的高斯消元法进行求解（参见正文）。

图17.3.1　在两端点给定边界值条件的线性有限差分方程的矩阵结构

```
1    X X                                   V    B
 1   X X                                   V    B
  1 X X                                    V    B
   1          X X                          V    B
    1         X X                          V    B
     1        X X                          V    B
      1       X X                          V    B
       1 X X                               V    B
        1           X X                    V    B
         1          X X                    V    B
          1         X X                    V    B
           1        X X                    V    B
            1 X X                          V    B
             1             X X             V    B
              1            X X             V    B
               1           X X             V    B
                1          X X             V    B
                 1 X X                     V    B
                  1                        V    B
                   1                       V    B
```

图17.3.1中的矩阵一旦化成这个结构以后，通过回代过程马上可以得出解。

图17.3.2　高斯消元法的目标结构

而且，整个求解过程，除了最后的回代过程以前，每次都只对矩阵的一块进行操作。这个求解过程包括四种类型的操作：(1)利用前一步的结果经约简后把部分元素化为零；(2)通过消元把剩下的块结构进行消元，让矩阵对角线上的元素为单位 1，非对角线上元素为 0；(3)把余下非零系数存储起来以备后用；(4)回代。我们用图一步一步地进行说明。

对于满足初始化边界条件，用以求解校正量的方程子块，我们有 $n_1$ 个方程，$N$ 个未知校正量。我们希望把第一子块进行变形，让子块的左边 $n_1\times n_1$ 的方阵变成沿对角线为单位元，而非对角线都为零的形式。图17.3.3说明了矩阵第一块最初和最后的形式。在图中，我们把需要进行对角化的元素标明为"$D$"，要发生变化的元素标明为"$A$"；在最后的块结构中，要存储起来的元素标明为"$S$"。我们依次选择前 $n_1$ 列的 $n_1$ 个"主元"，把主元所在的行归一化，让主元的值为单位一。然后，我们把该行适当的倍数加到其它行上去，使其它行中该主元所在列的元素都为零。在最后的形式中，约简的矩阵说明了，在网格点 1 的前 $n_1$ 个变元的校正量的值，是如何根据在网格点1处剩下的 $n_2$ 个未知校正量的值进行表示的，即，我们现在知道了前 $n_1$ 个分量是如何由其余 $n_2$ 个分量决定的。因此，我们只需把初始块中最后 $n_2$ 个非零的列以及矩阵方程右端向量对应的列存储起来。

```
(a)  D D D A        V  A
     D D D A A      V  A
     D D D A A      V  A

(b)  1 0 0 S S      V  S
     0 1 0 S S      V  S
     0 0 1 S S      V  S
```

(a)块的最初形式，(b)最终形式(参见正文解释)。

图17.3.3　对图7.3.1中矩阵的第一块(左上角)进行约简的过程

这里，我们需要强调一下这个方法的一个很重要的地方，在操作块时要使用较小存储量的时候，这一点是必须的，那就是导数矩阵 $S$ 的列的顺序应保证主元能在矩阵的前 $n_1$ 行找到，就是说，在第一个点的 $n_1$ 个边界条件必须和最初的 $j=1,2,\cdots n_1$ 个应变量 y[j][1]相关。如果不是的话，则第一块中最初的 $n_1\times n_1$ 方阵就有可能是奇异的，这种方法便失败了。或者，我们必须在块中所有 $N$ 列寻找主元，这就需要进行列的交换而需更多的记录工作。程序中提供了一个简便的方法重新对变量排序，例如对矩阵的列重新排序，这样，这一项工作很容易完成。这便是重要的细节。

其次，我们考虑表示有限差分方程组的 $N$ 个方程，这 $N$ 个方程描述了在点2和点1的2$N$ 个校正量之间的关系。该块中的元素以及前一步的结果都在图17.3.4中说明了。注意，加上第一块中行的适当的倍数，我们可以把该块中前 $n_1$ 列(标为"z")约简为零，而且这样做时我们只需要改变从第 $n_1+1$到 $N$ 列以及方程右端的元素。对于其它列我们可以把图中标有"$D$"的 $N\times N$ 方阵的元素对角化。最后，我们把图中标有"$A$"的 $n_2-1$列的元素进行了改变。图17.3.4中的第二部分表示我们对矩阵进行处理后的结果，其中标有"$S$"的$(n_2+1)\times N$ 个元素被存储起来。

我们在对下一个方程组，它对应于在点 3 和点 2 耦合的校正量的有限差分方程组，进行运算时，我们会发现，可用结果和新的方程的状态准确地重现前一段中所介绍的情况，这样我们可以依次进行哪些过程的计算，直到第 $M$ 子块。最后，在块 $M+1$ 上，我们又遇到了另外一个边界条件。

图17.3.5表示定义在网格点 $M$ 处，$N$ 个校正量的最后一块 $n_2$个有限常微分方程，以及以前块的约简结果。首先，我们仍用前面的结果把该块的前 $n_1$列化为零。在我们把剩下的方阵进行对角化的时候，我们即得到了有价值的结果：我们得出了在网格点 $M$ 处最终 $n_2$个的校正量值。

最后一子块约简以后，矩阵便为原先在图17.3.2表示的形式，矩阵即可以准备进行回代了。从最后一行开始，回推到最上面一行，在每一步，我们可以很简单地用已知量来决定未知校正量。

函数 **solvde** 把上面所述的各步组织起来。在算法中，我们所进行的基本过程是由函数 **solvde** 的内部调用函数来实现的。函数 **red** 把原来子块中 $S$ 矩阵的头几列进行约简，函数 **pinvs** 把 $S$ 矩阵中的各方阵对角化，并把未约简的系数存储起来。函数 **bksub** 实现往回替代的过程。作为函数 **solvde** 的用户必须了解所调用的参数，如下面所述，并提供一个被 **solvde** 调用的函数 **difeq**，用以计算矩阵 $S$ 各块的值。

```
(a)  1 0 0 S S              V  S
     0 1 0 S S              V  S
     0 0 1 S S              V  S
     Z Z Z D D D D D A A    V  A
     Z Z Z D D D D D A A    V  A
     Z Z Z D D D D D A A    V  A
     Z Z Z D D D D D A A    V  A
     Z Z Z D D D D D A A    V  A

(b)  1 0 0 S S              V  S
     0 1 0 S S              V  S
     0 0 1 S S              V  S
     0 0 0 1 0 0 0 0 S S    V  S
     0 0 0 0 1 0 0 0 S S    V  S
     0 0 0 0 0 1 0 0 S S    V  S
     0 0 0 0 0 0 1 0 S S    V  S
     0 0 0 0 0 0 0 1 S S    V  S
```

(a)原始形式，(b)最终形式(说明参见正文)。

图17.3.4　对图17.3.1中矩阵中间块的约简过程

```
(a)  0 0 0 1 0 0 0 0  S S   V  S
     0 0 0 0 1 0 0 0  S S   V  S
     0 0 0 0 0 1 0 0  S S   V  S
     0 0 0 0 0 0 1 0  S S   V  S
     0 0 0 0 0 0 0 1  S S   V  S
                  Z Z Z D D V  A
                  Z Z Z D D V  A
(b)  0 0 0 1 0 0 0 0  S S   V  S
     0 0 0 0 1 0 0 0  S S   V  S
     0 0 0 0 0 1 0 0  S S   V  S
     0 0 0 0 0 0 1 0  S S   V  S
     0 0 0 0 0 0 0 1  S S   V  S
                  0 0 0 1 0 V  S
                  0 0 0 0 1 V  S
```

(a)初始形式，(b)最终形式(说明参见正文)。

图17.3.5　对图17.3.1中最后一块(右下方)的约简过程

调用 **solvde** 的大多数变量都已经介绍过，但还需要一些讨论。矩阵 y[j][k] 包含解的初始预测值，以 j 标号在网格点 k 处的应变量。问题涉及到 k=1,…,m 个扩展点上的 ne 个有限差分方程。在第一个点 $k=1$ 提供了 nb 个边界条件，数组 indexv[j] 建立了 s 矩阵、方程(17.3.8)、(17.3.10)和(17.3.12)各列之间与应变量的关系，如上所述，用在 k=1 处的 nb 个边界条件来确定 s 矩阵前 nb 列所定义的应变量，这是很基本的一点。这样，用户能在程序 **difeq** 中排序 s 矩阵的第 j 列元素，以表示对应于应变量 indexv[j] 的导数。

在返回以前函数只准备进行 itmax 次校正循环，即使解尚未收敛也如此。参数 conv、slowc、scalv 和收敛有关。每次矩阵变换便产生在 m 个网点处 ne 个变元的校正量。我们希望随着迭代过程的继续，这些校正量逐渐趋于零，但我们必须定义一个校正量大小的量度。误差的“限额”是和各问题有关的，所以用户也许

想重写这部分程序代码。在下面的程序中，我们通过把所有校正量的绝对值加起来，并根据各种不同类型的变量加以不同的放大系数作为加权，来计算平均的校正误差"err"：

$$err = \frac{1}{m \times ne} \sum_{k=1}^{m} \sum_{j=1}^{ne} \frac{|\Delta Y[j][k]|}{scalv[j]} \qquad (17.3.13)$$

当 err ≤ conv 时，这种方法即收敛了。注意，用户需提供一个数组 scalv 用以度量各个变量的标准大小。

显然，如果 err 很大，我们偏离解的范围就很大，而且也许从一阶泰劳系数得出的校正量是准确的这想法本来就是错误的。slowc 数的大小调整了校正量的使用，在每次迭代以后，我们只应用了由矩阵求逆所求的校正量的一部分：

$$Y[j][k] \to Y[j][k] + \frac{slowc}{\max(slowc, err)} \Delta Y[j][k] \qquad (17.3.14)$$

这样，当 err > slowc 时，只用了调整量的一部分，但当 err ≤ slowc 时，整个调整量都使用上了。

调用语句也提供给 **solvde** 函数包含初始试验解的数组 y[1..yn][1..nyk]，以及工作空间数组 c[1..ne][1..ne−nb+1][1..m+1]，s[1..ne][1..2 * ne+1]。数组 c 是一个较大的数据块：它存储了为回代步骤所准备的还未约简的矩阵元素。如果有 m 个网点，就有 m+1 子块，每一子块又需要 ne 行和 ne−nb+1 列。尽管这个矩阵元素很多，但和未划分成块所需整个矩阵[ne×m]²个元素相比来说是很小的了。

我们现在介绍用户提供的函数 **difeq** 的工作过程，程序头如下：

```
void difeq(int k, int k1, int k2, int jsf, int is1, int isf,
     int indexv[], int ne , float * *s, float * *y);
```

从 **difeq** 传递给 **solvde** 的信息只是导数矩阵 s[1..ne][1..2 * ne+1]；所有其它参数都是输入给程序 **difeq** 的，而且不应被变动。k 表示当网格点或是块序号，k1，k2表示网格中第一个和最后一个点。如果 k=k1或 k>k2时，相应子块便要涉及在最初点和最终点的边界值条件；否则，相应子块只对和点 k−1，k 中变元有关的有限差分方程组起作用。

信息在数组 s[i][j]中存储的约定，遵循在方程(17.3.8)、(17.3.10)及(17.3.12)中所使用的：行 i 表示方程，列 j 表示解中相应变元的导数值。记住，每个方程无论是在一个或两个点，都是依赖于 ne 个应变量元。这样，j 能从 1 变到 ne 或是 2 * ne。每一点上应变量排列的列序必须和 indxv[j]中的列的顺序一致。这样，对于不在边界处的子块，第一列乘以 ΔY(l=indexv[1],k−1)，第 ne+1列乘以 ΔY(l=indexv[1],k)。is1和 isf 给出在 s 矩阵中，对于这一子块需要填充的开始和最后的行数。jsf 表示存储由等式(17.3.3)～(17.3.5)导出的差分方程 $E_{j,k}$的列数。这样，−s[i][jsf]即是矩阵右端的向量，之所以加上负号，是因为 **difeq** 提供的是实际的差分方程 $E_{j,k}$的值，而不是它的负值。注意，**solvde** 给 jsf 提供了一个值，这样恰好把该差分方程就放在 s 矩阵中所有导数后边的列中。因此，**difeq** 希望能寻找放入 s[i][j]中的值，其中 is1≤i≤isf，而且 1≤j≤jsf。

最后，s[1..ne][1..2 * ne+1]以及 y[1..nyj][1..nyk]给函数 **difeq** 提供了 s 的存储空间和该次迭代的解变量 y。怎么使用这个程序将在下节中举例说明。

下面的程序代码中的很多思想，出自于 Eggleton[1]。

```
#inlude <stdio.h>
#include <math.h>
#include "nrutil.h"

void solvde(int itmax, float conv, float slowc, float scalv[], int indexv [],
     int ne, int nb, int m, float * *y, float * * *c, float * *s)
```

用松驰法求解两点边界值问题的驱动程序。itmax 是最大迭代次数，conv 是收敛准则(参见正文)，slowc 控制每次迭代后实际应用的校正量，scalv[1..ne]包含每个应变量的标准化大小，用以加权计算误差。indexv[1..ne]列出了用于构造导数矩阵 s[1..ne][1..2 * ne+1]中变量的列序(在第一个网格点处的 nb 个边界条件必须和在 indexv 中列出的前 nb 个变量相关)。该问题在每点上涉及 ne 个方程，有 ne 个可调整的应变量。在第一个网格点上，有 nb 个边界值

条件。总共有 m 个网点。y[1..ne][1..m]是一个二维数组，它包含在每一网点处所有应变量的初始试测值。在每次迭代中，它被所有计算的校正量所更改。数组 c[1..ne][1..ne−nb+1][1..m+1]及 s 提供了松弛法中所需的存储空间。

```
{
    void bksub(int ne, int nb, int jf, int k1, int k2, float ***c);
    void difeq(int k, int k1, int k2, int jsf, int is1, int isf,
        int indexv[], int ne, float **s, float **y);
    void pinvs(int ie1, int ie2, int je1, int jsf, int jc1, int k,
        float ***c, float **s);
    void red(int iz1, int iz2, int jz1, int jz2, int jm1, int jm2, int jmf,
        int ic1, int jc1, int jcf, int kc, float ***c, float **s);
    int ic1,ic2,ic3,ic4,it,j,j1,j2,j3,j4,j5,j6,j7,j8,j9;
    int jc1,jcf,jv,k,k1,k2,km,kp,nvars,*kmax;
    float err,errj,fac,vmax,vz,*ermax;

    kmax=ivector(1,ne);
    ermax=vector(1,ne);
    k1=1;                                   设置行、列标志
    k2=m;
    nvars=ne*m;
    j1=1;
    j2=nb;
    j3=nb+1;
    j4=ne;
    j5=j4+j1;
    j6=j4+j2;
    j7=j4+j3;
    j8=j4+j4;
    j9=j8+j1;
    ic1=1;
    ic2=ne-nb;
    ic3=ic2+1;
    ic4=ne;
    jc1=1;
    jcf=ic3;
    for (it=1;it<=itmax;it++) {             初次迭代循环
        k=k1;                               在第一点的边界条件
        difeq(k,k1,k2,j9,ic3,ic4,indexv,ne,s,y);
        pinvs(ic3,ic4,j5,j9,jc1,k1,c,s);
        for (k=k1+1;k<=k2;k++) {            在所有成对点的有限差分方程
            kp=k-1;
            difeq(k,k1,k2,j9,ic1,ic4,indexv,ne,s,y);
            red(ic1,ic4,j1,j2,j3,j4,j9,ic3,jc1,jcf,kp,c,s);
            pinvs(ic1,ic4,j3,j9,jc1,k,c,s);
        }
        k=k2+1;                             最后的边界条件
        difeq(k,k1,k2,j9,ic1,ic2,indexv,ne,s,y);
        red(ic1,ic2,j5,j6,j7,j8,j9,ic3,jc1,jcf,k2,c,s);
        pinvs(ic1,ic2,j7,j9,jcf,k2+1,c,s);
        bksub(ne,nb,jcf,k1,k2,c);           回代
        err=0.0;
        for (j=1;j<=ne;j++) {               检查收敛性，累积平均误差
            jv=indexv[j];
            errj=vmax=0.0;
            km=0;
            for (k=k1;k<=k2;k++) {          对每个应变量，找出最大误差的点
                vz=fabs(c[jv][1][k]);
                if (vz > vmax) {
                    vmax=vz;
                    km=k;
                }
                errj += vz;
            }
            err += errj/scalv[j];           给各应变量加权
            ermax[j]=c[jv][1][km]/scalv[j];
```

```
            kmax[j]=km;
        }
        err /= nvars;
        fac=(err > slowc ? slowc/err : 1.0);  误差较大时，减小所用校正量

        for (j=1;j<=ne;j++) {                      应用校正量
            jv=indexv[j];
            for (k=k1;k<=k2;k++)
                y[j][k] -= fac*c[jv][1][k];
        }
        printf("\n%8s %9s %9s\n","Iter.","Error","FAC");    累积这一步的校正量
        printf("%6d %12.6f %11.6f\n",it,err,fac);
        if (err < conv) {
            free_vector(ermax,1,ne);
            free_ivector(kmax,1,ne);
            return;
        }
    }
    nrerror("Too many iterations in solvde");               收敛失败
}

void bksub(int ne, int nb, int jf, int k1, int k2, float ***c)      向后回代，在solvde中调用
{
    int nbf,im,kp,k,j,i;
    float xx;

    nbf=ne-nb;
    im=1;
    for (k=k2;k>=k1;k--) {                                 利用递归关系消去余下的依赖关系
        if (k == k1) im=nbf+1;
        kp=k+1;
        for (j=1;j<=nbf;j++) {
            xx=c[j][jf][kp];
            for (i=im;i<=ne;i++)
                c[i][jf][k] -= c[i][j][k]*xx;
        }
    }
    for (k=k1;k<=k2;k++) {                                 重排第一列中的校正量
        kp=k+1;
        for (i=1;i<=nb;i++) c[i][1][k]=c[i+nbf][jf][k];
        for (i=1;i<=nbf;i++) c[i+nb][1][k]=c[i][jf][kp];
    }
}

#include <math.h>
#include "nrutil.h"

void pinvs(int ie1, int ie2, int je1, int jsf, int jc1, int k, float ***c, float **s)
    把矩阵s中的子方阵对角化，并把c中的递归系数存储起来，在solvde中调用。
{
    int js1,jpiv,jp,je2,jcoff,j,irow,ipiv,id,icoff,i,*indxr;
    float pivinv,piv,dum,big,*pscl;

    indxr=ivector(ie1,ie2);
    pscl=vector(ie1,ie2);
    je2=je1+ie2-ie1;
    js1=je2+1;
    for (i=ie1;i<=ie2;i++) {              隐式选主元，如第2.1节
        big=0.0;
        for (j=je1;j<=je2;j++)
            if (fabs(s[i][j]) > big) big=fabs(s[i][j]);
        if (big == 0.0) nrerror("Singular matrix - row all 0, in pinvs");
        pscl[i]=1.0/big;
```

```
            indxr[i]=0;
        }
        for (id=ie1;id<=ie2;id++) {
            piv=0.0;
            for (i=ie1;i<=ie2;i++) {        寻找主元素
                if (indxr[i] == 0) {
                    big=0.0;
                    for (j=js1;j<=js2;j++) {
                        if (fabs(s[i][j]) > big) {
                            jp=j;
                            big=fabs(s[i][j]);
                        }
                    }
                    if (big*pscl[i] > piv) {
                        ipiv=i;
                        jpiv=jp;
                        piv=big*pscl[i];
                    }
                }
            }
            if (s[ipiv][jpiv] == 0.0) nrerror('Singular matrix in routine pinvs");
            indxr[ipiv]=jpiv;                       同时约化、保存已排序后的列
            pivinv=1.0/s[ipiv][jpiv];
            for (j=js1;j<=jsf;j++) s[ipiv][j] *= pivinv;      归一化主元行
            s[ipiv][jpiv]=1.0;
            for (i=ie1;i<=ie2;i++) {        约化列中的非主元素
                if (indxr[i] != jpiv) {
                    if (s[i][jpiv]) {
                        dum=s[i][jpiv];
                        for (j=js1;j<=jsf;j++)
                            s[i][j] -= dum*s[ipiv][j];
                        s[i][jpiv]=0.0;
                    }
                }
            }
        }
        jcoff=jc1-js1;                          排列并存储未约化的系数
        icoff=ie1-je1;
        for (i=ie1;i<=ie2;i++) {
            irow=indxr[i]+icoff;
            for (j=js1;j<=jsf;j++) c[irow][j+jcoff][k]=s[i][j];
        }
        free_vector(pscl,ie1,ie2);
        free_ivector(indxr,ie1,ie2);
}
```

void red(int iz1, int iz2, int jz1, int jz2, int jm1, int jm2, int jmf,
    int ic1, int jc1, int jcf, int kc, float ***c, float **s)

用存储在 c 矩阵的原先结果来简化 s 矩阵中第 jz1－jz2列，只有 jm1－jm2，jmf 列会受以前的结果影响，本程序 red 被 solvde 内部调用。

```
{
    int loff,l,j,ic,i;
    float vx;

    loff=jc1-jm1;
    ic=ic1;
    for (j=jz1;j<=jz2;j++) {                                    对要零化的列循环
        for (l=jm1;l<=jm2;l++) {                                对要变动的列循环
            vx=c[ic][l+loff][kc];
            for (i=iz1;i<=iz2;i++) s[i][l] -= s[i][j]*vx;       对行循环
        }
        vx=c[ic][jcf][kc];
        for (i=iz1;i<=iz2;i++) s[i][jmf] -= s[i][j]*vx;         加上最终元素。
        ic += 1;
    }
```

}

### 17.3.1 微分方程的"代数困难"集

松驰法允许利用的另一个便利之处，虽然不很明显，但对某些计算的速度能极大地提高。变量 $y_{i,k}$ 集没有必要直接和原始微分方程应变量相关，它们可以通过代数方程和那些变量联系。显然，解变量允许我们计算函数 $y$、$g$、$B$、$C$，而它们是用于从常微分方程来构造有限差分方程的，这是唯一必要的。在某些问题中，$g$ 依赖于隐式的 $y$ 函数，这样需要用迭代解法来计算常微分方程中的函数。通常情况下，可以通过定义一个新的变量集来处理这种"内部"非线性问题，从这个新的变量集中，我们可以直接得出 $y$、$g$ 和边界条件。典型的例子是在物理学中，其中要求一个复杂的状态方程的解。而在状态方程中，用其它变量而不是用常微分方程中的原始应变量的形式来表达会更加方便。这种近似过程，就类似于直接对原始的常微分方程进行了一次解析变换，这种解析也许是极为复杂的。但在松驰法中的变量替换很容易，而且不需要解析操作。

参考文献和进一步读物：


Eggleton, p. p1971, *Monthly Notices of the Royal Astronomical Society*, vol. 151, pp. 351～364 [1]


## 17.4 一个实例：球体调和函数

理解前一节算法的最好的方法是观看它们在实际问题中是如何使用的。作为范例，我们选择了球体调和函数的计算(更一般的名字是球体角函数，但我们更喜欢球体调和函数这种稳含说法)，我们会说明怎样找球体调和函数，首先采用松驰法(第17.3节)，然后用打靶法(第17.1节)和射向某一合适点(第17.2节)。

球体调和函数一般出现在，当在球体坐标系中，用变量分离法求解某种偏微分方程组的情况。它们在区间 $-1 \leqslant x \leqslant 1$ 上满足下面的微分方程：

$$\frac{d}{dx}\left[(1-x^2)\frac{dS}{dx}\right]+\left(\lambda-c^2x^2-\frac{m^2}{1-x^2}\right)S=0 \qquad (17.4.1)$$

这里 $m$ 是一个整数，$c$ 是"偏心率"，$\lambda$ 是特征值。另外，$c^2$ 可正可负，对于 $c^2>0$，函数叫作"长椭形"，而 $c^2<0$ 则称为"扁椭形"。方程在 $x=\pm1$ 处有奇异点，而且求解时，方程的边界值条件要在 $x=\pm1$ 处满足，只有对特征值 $\lambda$ 的某些取值，这种情况才可能。

我们首先考虑第一种球面的情况，$c=0$，我们把微分方程当作勒让德函数 $P_n^m(x)$。在这种情况下，特征值是 $\lambda_{mn}=n(n+1)$，$n=m,m-1,\cdots$。整数 $n$ 表示对固定 $m$ 值的连续特征值；当 $n=m$ 时，我们有最低的特征值，而且在区间 $-1<x<1$ 中相应的特征函数没有结点；当 $n=m+1$ 时，我们有下一个特征值，在$(-1,1)$中特征函数有一个结点；以此类推。

对于一般 $c^2\neq0$，有一种类似情况成立。我们把式(17.4.1)中特征值写作 $\lambda_{mn}(c)$，特征函数记作 $S_{mn}(x;c)$。对固定的 $m$，$n=m,m-1,\cdots$ 标明了相继的特征值。

一般 $\lambda_{mn}(c)$ 和 $S_{mn}(x;c)$ 的计算就很困难。关于复杂的递归关系式，幂级数展开等都可以在参考文献中找到[1-3]。通过对微分方程直接求解的简便计算很容易实现。

第一步是考查奇异点 $x=\pm1$ 附近的解的性态。在方程(17.4.1)中用幂级数展开式

$$S=(1\pm x)^{\alpha}\sum_{k=0}^{\infty}a_k(1\pm x)^k \qquad (17.4.2)$$

来替换，我们发现正则解有 $\alpha=m/2$。(不失一般性，我们可以取 $m\geqslant0$，因为方程是对称性的，

所以 $m$ 取正取负都可以。)我们把这个解提出公因子,即得到一个更容易数值法跟踪求解的方程。相应地,我们得到

$$S = (1-x^2)^{m/2}y \tag{17.4.3}$$

这样,我们由式(17.4.1)发现,$y$ 满足方程

$$(1-x^2)\frac{d^2y}{dx^2} - 2(m+1)x\frac{dy}{dx} + (\mu - c^2x^2)y = 0 \tag{17.4.4}$$

其中

$$\mu = \lambda - m(m+1) \tag{17.4.5}$$

在 $x \to -x$ 时,方程(17.4.1)和(17.4.5)都是不变的。因此函数 $S$ 和 $y$ 也必须是不变的,除非可能有一个放大因子(因为方程组是线性的,常数和解相乘以后仍是满足方程的解),因为解要被归一化,所以放大因子只可能是 ±1。如果($n-m$)是奇数,则在区间(-1,1)中,能有奇数个零点。因此我们必须选择一个反对称的解 $y(-x) = -y(x)$,它在 $x=0$ 处有一个零点;反之,如果 $n-m$ 是偶数,我们必须有对称解。这样,

$$y_{mn}(-x) = (-1)^{n-m}y_{mn}(x) \tag{17.4.6}$$

对 $S_{mn}$ 是类似的。

对式(17.4.4)的边界条件要求 $y$ 在 $x=\pm1$处应是正则的,换言之,在端点附近,解的形式应是

$$y = a_0 + a_1(1-x^2) + a_2(1-x^2)^2 + \cdots \tag{17.4.7}$$

把这个表达式代入方程(17.4.4)中,并让 $x \to 1$,我们发现

$$a_1 = -\frac{\mu - c^2}{4(m+1)}a_0 \tag{17.4.8}$$

等价地有

$$y'(1) = \frac{\mu - c^2}{2(m+1)}y(1) \tag{17.4.9}$$

在 $x=-1$处,一个类似方程成立而等号右边有一负号。在端点处,非正则解在函数和导数之间有不同关系。

我们可以利用式(17.4.6)的对称性,从零到 1 积分来代替方程从 -1 到 1 的积分。在 $x=0$处的边界条件为

$$\begin{aligned} y(0) &= 0, \quad n-m \text{ 为奇数} \\ y'(0) &= 0, \quad n-m \text{ 为偶数} \end{aligned} \tag{17.4.10}$$

第三个边界值条件来自这样一个事实:任一常数乘以某一解的结果仍是方程的解。这样我们可以对解进行归一化。我们采用这样的归一化,使在 $x=1$时,函数 $S_{mn}$和 $P_n^m$ 有同样的极限特性:

$$\lim_{x\to1}(1-x^2)^{-m/2}S_{mn}(x;c) = \lim_{x\to1}(1-x^2)^{-m/2}P_n^m(x) \tag{17.4.11}$$

各种归一化方法的约定都由 Flammer 列表进行了说明[1]。

对式(17.4.4)的二阶方程加上三个边界条件以后,问题即变成了求 $\lambda$ 的特征值问题,或等价于求 $\mu$ 的特征值问题。我们把它写成标准形式,只要设置

$$y_1 = y \tag{17.4.12}$$

$$y_2 = y' \tag{17.4.13}$$

$$y_3 = \mu \tag{17.4.14}$$

则

$$y_1' = y_2 \tag{17.4.15}$$

$$y_2' = \frac{1}{1-x^2}\left[2x(m+1)y_2 - (y_3 - c^2x^2)y_1\right] \tag{17.4.16}$$

$$y_3' = 0 \tag{17.4.17}$$

在这些符号表示中，$x=0$ 处的边界条件是

$$\begin{aligned} y_1 &= 0, \quad n-m \text{ 为奇数} \\ y_2 &= 0, \quad n-m \text{ 为偶数} \end{aligned} \tag{17.4.18}$$

在 $x=1$ 处，我们有两个条件：

$$y_2 = \frac{y_3 - c^2}{2(m+1)} y_1 \tag{17.4.19}$$

$$y_1 = \lim_{x\to 1}(1-x^2)^{-m/2}P_n^m(x) = \frac{(-1)^m(n+m)!}{2^m m!(n-m)!} \equiv \gamma \tag{17.4.20}$$

现在，我们必须决定用前面章节的算法来处理这个问题。

## 17.4.1 松驰法

如果我只需要几个 $\lambda$ 或 $S$ 的离散值，则打靶法也许是最快的方法。但是，如果我们需要一个较大的 $c$ 的序列值，则用松驰法比较好。松驰法的好处是一个较好的初始预测解能较快地收敛，而且如果 $c$ 变化较小时，先前求得的解应该是一个较好的初始预测解。

为简单起见，在区间 $0 \leqslant x \leqslant 1$ 中我们选择均匀的网格点。对总共 $M$ 个网格点，我们有

$$h = \frac{1}{M-1} \tag{17.4.21}$$

$$x_k = (k-1)h, \qquad k = 1,2,\ldots,M \tag{17.4.22}$$

在内点 $k=2,3,\cdots,M$ 处，方程(17.4.15)给出

$$E_{1,k} = y_{1,k} - y_{1,k-1} - \frac{h}{2}(y_{2,k} + y_{2,k-1}) \tag{17.4.23}$$

方程(17.4.16)给出

$$E_{2,k} = y_{2,k} - y_{2,k-1} - \beta_k \times \left[\frac{(x_k + x_{k-1})(m+1)(y_{2,k} + y_{2,k-1})}{2} - \alpha_k \frac{(y_{1,k} + y_{1,k-1})}{2}\right] \tag{17.4.24}$$

其中

$$\alpha_k = \frac{y_{3,k} + y_{3,k-1}}{2} - \frac{c^2(x_k + x_{k-1})^2}{4} \tag{17.4.25}$$

$$\beta_k = \frac{h}{1 - \frac{1}{4}(x_k + x_{k-1})^2} \tag{17.4.26}$$

最后，方程(17.4.17)给出

$$E_{3,k} = y_{3,k} - y_{3,k-1} \tag{17.4.27}$$

在上节中我们说明过，方程(17.3.8)中偏导数矩阵 $S_{i,j}$ 的定义，它用 $i$ 标明方程而 $j$ 标明变量。在这种情况下，在 $k-1$ 处，$y_j$ 中 $j$ 从 1 到 3，而在 $k$ 处，$y_j$ 从 4 到 6。因此，方程(17.4.23)给出

$$\begin{aligned} S_{1,1} &= -1, \quad S_{1,2} = -\frac{h}{2}, \quad S_{1,3} = 0 \\ S_{1,4} &= 1, \quad S_{1,5} = -\frac{h}{2}, \quad S_{1,6} = 0 \end{aligned} \tag{17.4.28}$$

同样，方程(17.4.24)给出

$$\begin{aligned} S_{2,1} &= \alpha_k\beta_k/2, & S_{2,2} &= -1 - \beta_k(x_k + x_{k-1})(m+1)/2, \\ S_{2,3} &= \beta_k(y_{1,k} + y_{1,k-1})/4, & S_{2,4} &= S_{2,1}, \\ S_{2,5} &= 2 - S_{2,2}, & S_{2,6} &= S_{2,3} \end{aligned} \tag{17.4.29}$$

而从方程(17.4.27)我们发现

$$\begin{aligned} S_{3,1} &= 0, \quad S_{3,2} = 0, \quad S_{3,3} = -1 \\ S_{3,4} &= 0, \quad S_{3,5} = 0, \quad S_{3,6} = 1 \end{aligned} \tag{17.4.30}$$

在 $x=0$，我们有边界值条件

$$E_{3,1} = \begin{cases} y_{1,1} & n-m \text{ 为奇数} \\ y_{2,1} & n-m \text{ 为偶数} \end{cases} \tag{17.4.31}$$

回忆在上节 **solvde** 程序中采用的约定，对 $k=1$ 处的一个边界条件，只有 $S_{3,j}$ 可以非零。同样，$j$ 只能取 4 到 6 的值，这是因为边界条件只和 $y_k$ 有关，而和 $y_{k-1}$ 无关。相应地，在 $x=0$ 处 $S_{3,j}$ 的唯一非零值是

$$\begin{aligned} S_{3,4} &= 1, \quad n-m \text{ 为奇数} \\ S_{3,5} &= 1, \quad n-m \text{ 为偶数} \end{aligned} \tag{17.4.32}$$

在 $x=1$ 处，我们有

$$E_{1,M+1} = y_{2,M} - \frac{y_{3,M} - c^2}{2(m+1)} y_{1,M} \tag{17.5.33}$$

$$E_{2,M+1} = y_{1,M} - \gamma \tag{17.4.34}$$

这样

$$S_{1,4} = -\frac{y_{3,M} - c^2}{2(m+1)}, \quad S_{1,5} = 1, \quad S_{1,6} = -\frac{y_{1,M}}{2(m-1)} \tag{17.4.35}$$

$$S_{2,4} = 1, \qquad S_{2,5} = 0, \quad S_{2,6} = 0 \tag{17.4.36}$$

现在我们给出实现上述算法的一个范例程序，我们需要一个主程序 **sfzoid**，它调用 **solvde** 程序，而且我们还得提供一个由 **solvde** 调用的 **difeq**。为简单起见，我们选择用 $M=41$ 个网格点的等距划分的网格，也就是说，$h=0.025$。正如我们将会看到的那样，这对特征值以及 $n-m$ 个调整值等都能保证很好的精确度。

因为如果 $n-m$ 为偶数的话，在 $x=0$ 处的边界条件和 $y_1$ 无关，所以我们必须用 solvde 中的 indexv 特性。我们介绍过，indexv[j]描述了在 s[i][j]的哪一列中放入了变量 y[j]。如果 $n-m$ 是偶数，则我们必须把 $y_1$ 和 $y_2$ 的列相互对调，这样使 s[i][j]中没有零主元。

程序提示输入 $m$ 和 $n$ 的值，然后计算出满足勒让德函数 $P_n^m$ 的 $y$ 的初始预测值，其次提示输入 $c^2$，解出 $y$，再提示输入 $c^2$，用先前的值作为初始预测值解出 $y$，这样循环下去。

```
#include <stdio.h>
#include <math.h>
#include "nrutil.h"
#define NE 3
#define M 41
#define NB 1
#define NSI NE
#define NYJ NE
#define NYK M
#define NCI NE
#define NCJ (NE-NB+1)
#define NCK (M+1)
#define NSJ (2*NE+1)

int mm, n, mpt=M;
float h, c2=0.0, anorm, x[M+1];                    与程序 difeq 通信的全局变量

int main(void) /* 程序 sfroid */
    使用 solvde 的范例程序,计算 m≥0和 n≥m 的球体调和函数 S_mn(x,c)的特征值。在程序中,m 是 mm,c²是 c2,方程
    (17.4.20)中的 γ 是 anorm。
{
    float plgndr(int l, int m, float x);
    void solvde(int itmax, float conv, float slowc, float scalv[],
        int indexv[], int ne, int nb, int m, float **y, float ***c, float **s);
    int i,itmax,k,indexv[NE+1];
    float conv,deriv,fac1,fac2,q1,slowc,scalv[NE+1];
    float **y,**s,***c;

    y=matrix(1,NYJ,1,NYK);
    s=matrix(1,NSI,1,NSJ);
    c=f3tensor(1,NCI,1,NCJ,1,NCK);
    itmax=100;
    conv=5.0e-6;
    slowc=1.0;
    h=1.0/(M-1);
    printf("\nenter m n\n");
    scanf("%d %d",&mm,&n);
    if (n+mm & 1) {                                    没必要调换
        indexv[1]=1;
        indexv[2]=2;
        indexv[3]=3;
    } else {                                           调换 y1 和 y2
        indexv[1]=2;
        indexv[2]=1;
        indexv[3]=3;
    }
    anorm=1.0;                                         计算 γ
    if (mm) {
        q1=n;
          for (i=1;i<=mm;i++) anorm = -0.5*anorm*(n+i)*(q1--/i);
      }
      for (k=1;k<=(M-1);k++) {                         初始猜测值
          x[k]=(k-1)*h;
          fac1=1.0-x[k]*x[k];
          fac2=exp((-mm/2.0)*log(fac1));
          y[1][k]=plgndr(n,mm,x[k])*fac2;              P_n^m 是第6.6节中的
          deriv = -((n-mm+1)*plgndr(n+1,mm,x[k])-      从递归关系式求 P_n^m 导数
              (n+1)*x[k]*plgndr(n,mm,x[k]))/fac1;
          y[2][k]=mm*x[k]*y[1][k]/fac1+deriv*fac2;
          y[3][k]=n*(n+1)-mm*(mm+1);
      }
      x[M]=1.0;                                        在 x=1 处的初始猜测值已单独给出
      y[1][M]=anorm;
```

```
    y[3][M]=n*(n+1)-mm*(mm+1);
    y[2][M]=(y[3][M]-c2)*y[1][M]/(2.0*(mm+1.0));
    scalv[1]=fabs(anorm);
    scalv[2]=(y[2][M] > scalv[1] ? y[2][M] : scalv[1]);
    scalv[3]=(y[3][M] > 1.0 ? y[3][M] : 1.0);
    for (;;) {
        printf("\nEnter c**2 or 999 to end.\n");
        scanf("%f",&c2);
        if (c2 == 999) {
            free_f3tensor(c,1,NCI,1,NCJ,1,NCK);
            free_matrix(s,1,NSI,1,NSJ);
            free_matrix(y,1,NYJ,1,NYK);
            return 0;
        }
        solvde(itmax,conv,slowc,scalv,indexv,NE,NB,M,y,c,s);
        printf("\n %s %2d %s %2d %s %7.3f %s %10.6f\n",
            "m =",mm," n =",n," c**2 =",c2,
            " lamda =",y[3][1]+mm*(mm+1));
    }                                                   返回c^2的另一值
}
```

```
extern int mm,n,mpt;                                    在 sfroid中定义
extern float h,c2,anorm,x[];

void difeq(int k, int k1, int k2, int jsf, int is1, int isf, int indexv[],
    int ne, float **s, float **y)
为solvde返回矩阵 s
{
    float temp,temp1,temp2;

    if (k == k1) {                                      Boundary condition at first point.
        if (n+mm & 1) {
            s[3][3+indexv[1]]=1.0;                      方程(17.4.32)
            s[3][3+indexv[2]]=0.0;
            s[3][3+indexv[3]]=0.0;
            s[3][jsf]=y[1][1];                          方程(17.4.31)
        } else {
            s[3][3+indexv[1]]=0.0;                      方程(17.4.32)
            s[3][3+indexv[2]]=1.0;
            s[3][3+indexv[3]]=0.0;
            s[3][jsf]=y[2][1];                          方程(17.4.31)
        }
    } else if (k > k2) {                                在末点的边界值
        s[1][3+indexv[1]] = -(y[3][mpt]-c2)/(2.0*(mm+1.0));          (17.4.35)
        s[1][3+indexv[2]]=1.0;
        s[1][3+indexv[3]] = -y[1][mpt]/(2.0*(mm+1.0));
        s[1][jsf]=y[2][mpt]-(y[3][mpt]-c2)*y[1][mpt]/(2.0*(mm+1.0));  (17.4.33)
        s[2][3+indexv[1]]=1.0;                          方程(17.4.36)
        s[2][3+indexv[2]]=0.0;
        s[2][3+indexv[3]]=0.0;
        s[2][jsf]=y[1][mpt]-anorm;                      方程(17.4.34)
    } else {                                            内点.
        s[1][indexv[1]] = -1.0;                         方程(17.4.28)
        s[1][indexv[2]] = -0.5*h;
        s[1][indexv[3]]=0.0;
        s[1][3+indexv[1]]=1.0;
        s[1][3+indexv[2]] = -0.5*h;
        s[1][3+indexv[3]]=0.0;
        temp1=x[k]+x[k-1];
        temp=h/(1.0-temp1*temp1*0.25);
        temp2=0.5*(y[3][k]+y[3][k-1])-c2*0.25*temp1*temp1;
        s[2][indexv[1]]=temp*temp2*0.5;                 方程(17.4.29)
        s[2][indexv[2]] = -1.0-0.5*temp*(mm+1.0)*temp1;
        s[2][indexv[3]]=0.25*temp*(y[1][k]+y[1][k-1]);
```

```
        s[2][3+indexv[1]]=s[2][indexv[1]];
        s[2][3+indexv[2]]=2.0+s[2][indexv[2]];
        s[2][3+indexv[3]]=s[2][indexv[3]];
        s[3][indexv[1]]=0.0;                        方程(17.4.30)
        s[3][indexv[2]]=0.0;
        s[3][indexv[3]] = -1.0;
        s[3][3+indexv[1]]=0.0;
        s[3][3+indexv[2]]=0.0;
        s[3][3+indexv[3]]=1.0;
        s[1][jsf]=y[1][k]-y[1][k-1]-0.5*h*(y[2][k]+y[2][k-1]);      方程(17.4.23)
        s[2][jsf]=y[2][k]-y[2][k-1]-temp*((x[k]+x[k-1])             方程(17.4.24)
            *0.5*(mm+1.0)*(y[2][k]+y[2][k-1])-temp2
            *0.5*(y[1][k]+y[1][k-1]));
        s[3][jsf]=y[3][k]-y[3][k-1];                 方程(17.4.27)
    }
}
```

读者可以运行这个程序，然后和 Flammer 书[2]中的表，或是在 Abramowitz 和 Stegun 书[4]中表21.1给出的 $\lambda_{mn}(c)$的值进行对照。通常情况下，大约 3 次迭代后就收敛。表17.4.1给出了一些比较值。

**表17.4.1　从 sfroid 中选择的输出结果**

| $m$ | $n$ | $c^2$ | $\lambda_{exact}$ | $\lambda_{sfroid}$ |
|---|---|---|---|---|
| 2 | 2 | 0.1 | 6.01427 | 6.01427 |
| | | 1.0 | 6.14095 | 6.14095 |
| | | 4.0 | 6.54250 | 6.54253 |
| 2 | 5 | 1.0 | 30.4361 | 30.4372 |
| | | 16.0 | 36.9963 | 37.0135 |
| 4 | 11 | −1.0 | 131.560 | 131.554 |

## 17.4.2　打靶法

为用打靶法(第17.4节)解决同样的问题，我们提供了一个函数 **derivs** 来实现等式(17.4.5)～(17.4.17)。我们将在区间 $-1\leqslant x\leqslant 0$ 上积分该方程组。我们提供了函数 **load** 把特征值 $y_3$设置为当前最佳估计值 v[1]。而且用等式(17.4.20)和(17.4.19)设置 $y_1$和 $y_2$的边界值(有一个负号，相应于 $x=-1$)。注意，边界条件实际上提供了一个到边界的距离 $dx$，以避免计算边界右边的 $y_2'$。函数 **scose** 来自于式(17.4.18)。

```
#include <stdio.h>
#include "nrutil.h"
#define N2 1

int m,n;                              与程序 load,score 和 derivs 传递的参数
float c2,dx,gmma;

int nvar;                             和 shoot 传递的参数
float x1,x2;

int main(void) /* Program sphoot */
    使用打靶法的范例程序。对 m≥0和 n≥m 的情况计算球体调和函数的 Smn(x;c)的特征值，注意 shoot(第17.1节)是
```

怎样为 **newt** 提供程序 vecfunc 的。

```
{
    void newt(float x[], int n, int *check,
        void (*vecfunc)(int, float [], float []));
    void shoot(int n, float v[], float f[]);
    int check,i;
    float q1,*v;

    v=vector(1,N2);
    dx=1.0e-4;                          避免在x=1处求导数
    nvar=3;                             方程个数
    for (;;) {
        printf("input m,n,c-squared\n");
        if (scanf("%d %d %f",&m,&n,&c2) == EOF) break;
        if (n < m || m < 0) continue;
        gmma=1.0;                       计算方程(17.4.20)的γ
        q1=n;
        for (i=1;i<=m;i++) gmma *= -0.5*(n+i)*(q1--/i);
        v[1]=n*(n+1)-m*(m+1)+c2/2.0;    特征值的初始猜测值
        x1 = -1.0+dx;                   设积分区域
        x2=0.0;
        newt(v,N2,&check,shoot);        找出使 score 中函数 f 为零的 v
        if (check) {
            printf("shoot failed; bad initial guess\n");
        } else {
            printf("\tmu(m,n)\n");
            printf("%12.6f\n",v[1]);
        }
    }
    free_vector(v,1,N2);
    return 0;
}

void load (float x1, float v[], float y[])      提供在 x=-1+dx 处的积分初始值
{
    float y1 = (n-m & 1 ? -gmma : gmma);
    y[3]=v[1];
    y[2] = -(y[3]-c2)*y1/(2*(m+1));
    y[1]=y1-y[2]*dx;
}

void score (float xf, float y[], float f[])     检测边界条件在 x=0处是否满足
{
    f[1]=(n-m & 1 ? y[1] : y[2]);
}

void derivs (float x, float y[], float dydx[])  为 odeint 计算导数值
{
    dydx[1]=y[2];
    dydx[2]=(2.0*x*(m+1.0)*y[2]-(y[3]-c2*x*x)*y[1])/(1.0-x*x);
    dydx[3]=0.0;
}
```

## 17.4.3 射向某一合适点

为全面起见，我们说明在第17.2节中的程序 **shootf**，其积分域为 $-1+dx \leqslant x \leqslant 1-dx$，而合适点选在 $x=0$ 处。程序 **derivs** 和 **shoot** 的调用一样。现在有两个 **load** 程序。对 $x=-1$ 程序 **load1**和上面的 **load** 大致相同，在 $x=1$ 处，**load2** 设置函数值 $y_1$和特征值 $y_3$为最佳当前值，分别为 v2[1]和 v2[2]。能够十分合理地使初始的特征猜测值在两个区间一样，则在

迭代过程中 v1[1]和 v2[2]。如果将保持相等。函数 **score** 用来简单地检查三个函数值在合适点是否相等。

```
#include <stdio.h>
#include <math.h>
#include "nrutil.h"
#define N1  2
#define N2  1
#define NTOT (N1+N2)
#define DXX 1.0e-4

int m,n;                        与程序 load1,load2,score,derive 传递的参数
float c2,dx,gmma;

int nn2,nvar;                   与程序 shootf 传递的参数
float x1,x2,xf;

int main (void)/* program sphfpt */
    使用 shootf 的范例程序：在 m≥0 和 n≥m 条件下计算球体调和函数 S_mn(x;c)的特征值。注意 shootf(第17.2节)是
    如何为 newt 提供程序 vecfunc 的，程序 derivs 和上面的一样。
{
    void newt(float x[], int n, int *check,
        void (*vecfunc)(int, float [], float []));
    void shootf(int n, float v[], float f[]);
    int check,i;
    float q1,*v1,*v2,*v;

    v=vector(1,NTOT);
    v1=v;
    v2 = &v[N2];
    nvar=NTOT;                              方程数目
    nn2=N2;
    dx=DXX;                                 避免在x=±1处求导
    for (;;) {
        printf("input m,n,c-squared\n");
        if (scanf("%d %d %f",&m,&n,&c2) == EOF) break;
        if (n < m || m < 0) continue;
        gmma=1.0;                           计算(17.4.20)的γ
        q1=n;
        for (i=1;i<=m;i++) gmma *= -0.5*(n+i)*(q1--/i);
        v1[1]=n*(n+1)-m*(m+1)+c2/2.0;      初始化特征值和函数值的猜测值
        v2[2]=v1[1];
        v2[1]=gmma*(1.0-(v2[2]-c2)*dx/(2*(m+1)));
        x1 = -1.0+dx;                       设积分域
        x2=1.0-dx;
        xf=0.0;                             拟合点
        newt(v,NTOT,&check,shootf);         找出在 score 中使函数 f 为零的 v
        if (check) {
            printf("shootf failed; bad initial guess\n");
        } else {
            printf("\tmu(m,n)\n");
            printf("%12.6f\n",v[1]);
        }
    }
    free_vector(v,1,NTOT);
    return 0;
}

void load1 (float x1, float v1[], float y[])       提供在 x=-1+dx 处的积分初始值
{
    float y1 = (n-m & 1 ? -gmma : gmma);
    y[3]=v1[1];
```

```
    y[2]= -(y[3]-c2)*y1/(2*(m+1));
    y[1]=y1+y[2]*dx;
}

void load2 (float x2, float v2[], float y[])          提供在 x=1-dx 处的积分初始值
{
    y[3]=v2[2];
    y[1]=v2[1];
    y[2]=(y[3]-c2)*y[1]/(2*(m-1));
}

void score (float xf, float y[], float f[])           检测解在合适点 x=0处是否相等
{
    int j;

    for (i=1;i<=3;i++) f[i]=y[i];
}
```

参考文献和进一步读物：


Flammer, C. 1957, *Spheroidal Wave Functions* (Stanford, CA:Stanford University Press). [1]

Abramowitz, M., and Stegun, I. A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, vol. 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §21. [2]

Morse, P. M., and Feshbach, H. 1953, *Methods of Theoretical Physics*, Part II (New York: McGraw-Hill), pp. 1502ff. [3]


## 17.5 网格点的自动分配

在松驰法问题中，我们需要选择在网点处自变量的值，这称作网格点的**分配**。通常是选用一些似乎合理的值作为其值。如果可行的话，则这组值满意；如果不行的话，增加网格点的数目，通常就可解决这个问题。

如果我们事先知道解在什么地方变化得较快，则可以在那儿多设几个网格点，而在其它地方少设几个。或者我们可以先用均匀的网格先求出解，然后看一看，我们应该在哪里多加一些点，接着我们用改进后的网格点重复求解过程，其目的都是为了设置网格点，以使尽可能地精确地表示解。

同样，自动进行网格点的分配也是可能的，这样，即可以在松驰过程中"动态地"进行求解。这种效用很大的技巧不只是改变了松驰法的求解精度，而且(正如我们下一节要看到的)可以用很漂亮的方法解决内部奇异性的问题，下面我们学习自动分配网格点是如何实现的。

我们把注意力集中于自变量 $x$，以及考虑两种交替的再次参数化方法。首先，我们给出 $q$，这正是对应于网格点本身的坐标，这样在 $k=1$ 处，$q=1$，在 $k=2$ 处，$q=2$；以此类推。在任意两个网格点之间，我们有 $\Delta q=1$。在常微分程中自变量 $x$ 变为 $q$ 后。

$$\frac{d\mathbf{y}}{dx}=\mathbf{g} \tag{17.5.1}$$

变成了

$$\frac{d\mathbf{y}}{dq}=\mathbf{g}\frac{dx}{dq} \tag{17.5.2}$$

按 $q$ 的形式，方程(17.5.2)作为有限差分方程可以写成

$$\mathbf{y}_k-\mathbf{y}_{k-1}-\frac{1}{2}\left[\left(\mathbf{g}\frac{dx}{dq}\right)_k+\left(\mathbf{g}\frac{dx}{dq}\right)_{k-1}\right]=0 \tag{17.5.3}$$

或其它相关的表达式,注意 $dx/dq$ 应该和 g 在一起,$x$ 和 $q$ 之间的变换只依赖于雅可比系数 $dx/dq$,其倒数 $dq/dx$ 和网格点密度成正比。

现在,若给出函数 $y(x)$,或在当前松弛步骤的近似式,则我们便能对怎样决定网格点的密度有了一些主意。例如,可能希望在 $y$ 变化较快之处,或是在边界之处,将 $dq/dx$ 的值取得大一些。事实上,我们也许可以构造这样一个式子,让 $dq/dx$ 和它们成正比,但问题是我们不知道这个比例常数,也就是说,我们构造的式子在 $x$ 的取值范围内——根据其定义,使 $q$ 从1变到 $M$——可能没有正确的积分。为了解决这个问题,我们引入了又一次的再参数化 $Q(q)$,其中 $Q$ 是一个新的自变量,$Q$ 和 $q$ 之间的关系取为线性的,这样网格划分间距公式对 $dQ/dx$ 来说,不同的只是其未知的比例常数。

由一个线性方程可知

$$\frac{d^2Q}{dq^2} = 0 \tag{17.5.4}$$

或是用通常的联立一阶方程组

$$\frac{dQ(x)}{dq} = \psi \qquad \frac{d\psi}{dq} = 0 \tag{17.5.5}$$

其中,$\psi$ 是一个新的中间变量。我们把这两个方程加到求解的常微分方程组中。

完成变量描述以后,我们加第三个常微分方程,也就是我们所需要的网格点密度函数即

$$\varphi(x) = \frac{dQ}{dx} = \frac{dQ}{dq}\frac{dq}{dx} \tag{17.5.6}$$

其中,$\varphi(x)$ 由我们选定,若按网格变量 $q$ 改写这个方程,即为

$$\frac{dx}{dq} = \frac{\psi}{\varphi(x)} \tag{17.5.7}$$

注意,$\varphi(x)$ 应该选为正定的,以使网点密度在各处都是正的,否则式(17.5.7)中分母可能为零。

用自动网格分划,把(17.5.5)~(17.5.7)这三个常微分方程加入方程组中,如加到数组 y[j][k]中。现在 $x$ 变成了一个应变量!Q 和 $\psi$ 也变成新的应变量。通常,计算 $\varphi$ 不需要多少额外工作量,因为它可以从已有 $g$ 的那些部分中构成。自动网格分划过程允许用户先看一下,数值结果会如何受各种网格点分划方法的影响(如果网格间距函数 Q 能通过解析得到,这是一种特例,即 $dQ/dx$ 直接可积。这样,只需额外加上两个方程,也就是式(17.5.5)中的两个方程,以及两个新的变量 $x$,$\psi$)。

作为实现这种方法的一种典型范例,我们考虑一个应变量 $y(x)$ 的系统。我们可以设置

$$dQ = \frac{dx}{\Delta} + \frac{|d\ln y|}{\delta} \tag{17.5.8}$$

或是

$$\varphi(x) = \frac{dQ}{dx} = \frac{1}{\Delta} + \left|\frac{dy/dx}{y\delta}\right| \tag{17.5.9}$$

其中 $\Delta$ 和 $\delta$ 是我们选择的常数。如果只有第一项,则它给出关于 $x$ 的均匀分划。而第二项则是使在 $y$ 迅速变化的地方使用更多的网格点,常数的作用是使 y 的每个对数变化量 $\delta$ 对在 $x$ 网格点变化量 $\Delta$ 有相当的影响。可以通过试验来调整这些常数。当然其它方法也是可以的,诸如可用 x 的对数分划,也就是在第一项中用 $d\ln x$ 来代替 $dx$。

## 17.6 内部边界条件或奇异点的处理

奇异性可能会出现在两点边界值问题的内部,典型地,假设有一个点 $x_s$,在该点的导数必须用下面的表达式计算

$$S(x_s) = \frac{N(x_s, \mathbf{y})}{D(x_s, \mathbf{y})} \tag{17.6.1}$$

其中分母 $D(x_s,y)=0$。在有限解的物理问题中，奇异点的问题通常伴随着自身可以解决的对策：当 $D\to0$ 时，物理解 $y$ 必须同时让 $N\to0$，这样，这个比例式便取有意义的值了。对解 $y$ 的限制通常称作**正则性条件**。使 $D(x_s,y)$ 在 $x_s$ 点满足某些特殊限制的条件，完全类似于一个额外的边界条件，即在某一点处应变量必须满足一个代数关系。

在第17.2节中，我们介绍了用"合适点方法"来处理，边界条件是奇异的方程内积分问题。我们也曾讨论过这个相关的问题。在那些问题中，无法从积分域的一端积分到另一端，但是常微分方程在奇异点附近确实有很好特性的导数和解，所以可以从丢弃该点进行积分。松弛法和"射向某一合适点"的方法解决这个问题都很容易。同样，在那些问题中，奇异特性的存在隔离了需要被满足进行求解的某些特殊的边界值。

这里所不同的是，我们考虑的奇异性是出现在中间点上，奇异点的位置和解有关系，所以事先是未知的。因此，我们面对一个循环的问题：奇异性妨碍我们寻求数值解法，而我们又需要用数值解法寻找奇异点的位置，这样的奇异性也和给某个变量选择一特殊值有关，这些变量允许解在奇异点处满足正则性条件。这样，内部奇异性的问题便采用了内部边界条件的问题。

处理内部奇异性的一种方法是，把这个问题当作一个自由边界值问题，象我们在第17.0节末尾讨论的那样。作为一个简单的例子，我们考虑方程

$$\frac{dy}{dx}=\frac{N(x,y)}{D(x,y)} \tag{17.6.2}$$

其中 $N$ 和 $D$ 都需要在某一未知点 $x_s$ 穿过零点。我们增加方程

$$z=x_s-x_1 \qquad \frac{dz}{dx}=0 \tag{17.6.3}$$

其中 $x_s$ 是未知的奇异点位置，并且通过下面设置把自变量改成 $t$。

$$x-x_1=tz \qquad 0\leq t\leq 1 \tag{17.6.4}$$

在 $t=1$ 处的边界条件变成

$$N(x,y)=0 \qquad D(x,y)=0 \tag{17.6.5}$$

另一种方法是，用前一节讨论的自适应网格点控制来处理内部奇异性的问题。对于式(17.6.2)中的问题，我们增加网格点分划方程

$$\frac{dQ}{dq}=\psi \tag{17.6.6}$$

$$\frac{d\psi}{dq}=0 \tag{17.6.7}$$

增加一个简单的网格点分划函数，它把 $x$ 直接映射到 $q$，其中 $q$ 从 1 变到 $M$。网格点的数目为

$$Q(x)=x-x_1 \qquad \frac{dQ}{dx}=1 \tag{17.6.8}$$

加入这三个一阶微分方程以后，我们还必须加入相应的边界条件。如果没有奇异性的话，这些将简便地是：

$$\text{在}\quad q=1:\quad x=x_1,\quad Q=0 \tag{17.6.9}$$

$$\text{在}\quad q=M:\quad x=x_2 \tag{17.6.10}$$

以及在 $q=1$ 处指定的总共 $N$ 个值 $y_i$。在这种情况下，问题变成一个在 $x_1$ 处指定了所有边界值条件的初始值问题，并且网格分划函数已是多余的。

但实际上，我们眼下要处理的条件是：

$$\text{在}\quad q=1:\quad x=x_1,\quad Q=0 \tag{17.6.11}$$

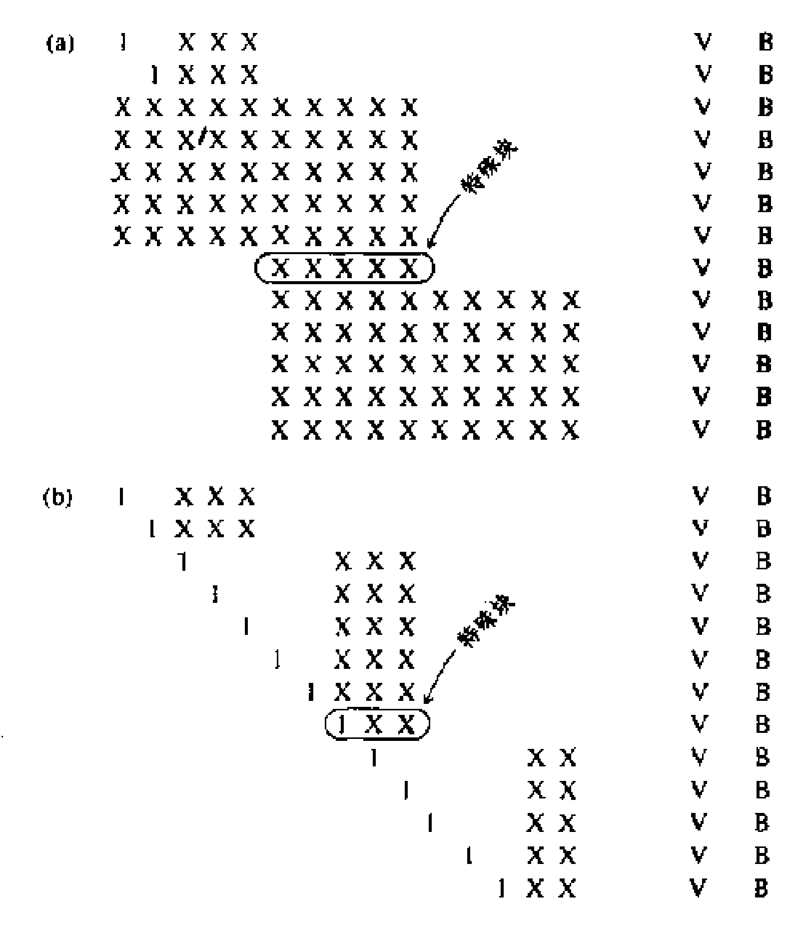
$$\text{在}\quad q=M:\quad N(x,y)=0,\quad Q(x,y)=0 \tag{17.6.12}$$

以及在 $q=1$ 处的 $N-1$ 个 $y_i$ 的值。"丢失"的那个 $y_i$ 值是要进行调整的，换言之，应如此调整，使得解以正则形式(零除零型)，而不是以非正则形式(有限值除零)经过奇异点。还应注意，这些边界条件并不直接给 $x_2$ 强制定义一个解，而使 $x_2$ 变成一个可调整的参数，不断变化以匹配正则性条件。

在这个例子中，奇异性出现在一个边界处，因为边界位置是未知的，从而增加了复杂度。在上述的例子

中，我们可以简单地把常微分方程积分到奇异点，然后作为另一个单独的问题，重新从奇异点开始积分到我们需要的地方。但是，当其它的奇异点出现在内部的情况中，并不能完全确定问题的全部所在；沿网格点还有某些更多的边界条件需要进一步满足。这些情况在原理上并不困难，但对第17.3节中所给出的松驰法程序代码确实需要一些改变。实际上所需要做的是，在内部边界条件出现的网格点处，加一块"特殊"的方程块，然后作一些适当的记录工作。

图17.6.1说明了一个实例，其中整个问题包含5个方程，在第一点有两个边界条件，一个"内在"边界条件和两个最终边界条件。这个图说明了矩阵中，特殊子块附近沿对角线的整个矩阵的结构。在区域的中间，一般子块中有5个方程（行）和10个未知量（列）。对于特殊子块前面的每一块，初始条件提供了足够的信息，把块的头两列化成零。五个有限常微分方程至少除去了多于五列，最后的三列存储起来进行回代（如第17.3节所示）。为了要处理额外的条件，我们跳出正常循环，并增加一个只有一个方程的特殊子块：内部边界条件。这极有效地降低了一列未约简的其余网格点系数的存储量，并且允许我们把以后块中的头三列约简成零。函数 **red**、**pinvs**、**bksub** 只需作少量的重新编码就能容易处理这些情况，但每个问题只针对一种特殊情况，因此用户必须根据需要自己来作修改。




内部条件引入了一个特殊块，(a)初始形式，和图17.3.1比较；(b)最终的形式，和图17.3.2比较，

图17.6.1　加上内部边界条件的有限差分方程

参考文献和进一步读物：


Lonoon, R.A., and Flannery, B.P.1982, *The Astrophysical Journal*, vol.258, pp. 260～269

# 第十八章　积分方程和反演理论

## 18.0　引言

许多人，除具备数值知识和人以外，都以为积分方程的数值求解是个极其神秘的课题，因此，迄今为止，数值分析教科书中几乎都没有包含此内容。实际上，积分方程数值求解方面的文献很多，目前还在不断地增加，并且也已出版了几本专著。所以单独成册出版的一个原因是，因为存在许多不同类型的积分方程，而且每类都有许多不同的、可能隐含的陷阱，即便是一个简单的情况也需要多种不同的算法进行处理。

线性积分方程和简单而又古老的线性方程之间有着紧密的对应关系，前者是在一个无限维函数空间内，函数间确定了一种线性积分关系；而后者则是在一个有限维向量空间内，向量间确定了一种线性关系。因为这种对应关系是大部分计算算法的核心，所以，在我们讨论积分方程的分类前，有必要进一步明确这一概念。

**弗雷德霍姆**(Fredholm)方程涉及具有固定上、下限的定积分。第一类非齐次弗雷德霍姆方程其形式如下

$$g(t) = \int_a^b K(t,s) f(s) ds \tag{18.0.1}$$

这里，$f(t)$是一个待求的未知函数，而$g(t)$是一个已知的"右端项"。(在积分方程中，由于某些奇怪的原因，熟知的"右端项"习惯上却写在左边！)有两个变量的函数$K(t,s)$称为**核**。方程(18.0.1)对应的矩阵方程为

$$\mathbf{K} \cdot \mathbf{f} = \mathbf{g} \tag{18.0.2}$$

其解为$\mathbf{f} = \mathbf{K}^{-1} \cdot \mathbf{g}$，而$\mathbf{K}^{-1}$是逆矩阵。与方程(18.0.2)类似，当$g$非零($g=0$的齐次情形几乎毫无用处)且$K$可逆时，方程(18.0.1)就有唯一解。但是，我们将看到，后一条件正如该规则一样常常出现例外。

类似于有限维特征值问题，

$$(\mathbf{K} - \sigma \mathbf{1}) \cdot \mathbf{f} = \mathbf{g} \tag{18.0.3}$$

叫做**第二类弗雷德霍姆(Fredholm)方程**，通常写为

$$f(t) = \lambda \int_a^b K(t,s) f(s) ds + g(t) \tag{18.0.4}$$

同样，符号的习惯并不严格对应：方程(18.0.4)中的$\lambda$对应着(18.0.3)中的$1/\sigma$，而$\mathbf{g}$对应着$-g/\lambda$。如果$g$(或$\mathbf{g}$)为零，则称方程是齐次的。如果核$K(t,s)$有界，则同方程(18.0.3)一样，方程(18.0.4)有这种性质，即其齐次形式在具有最多可数无限集$\lambda = \lambda_n, n=1,2\cdots$个**特征值**上有解。相应的解$f_n(t)$为**特征函数**。当核对称时特征值为实数。

在$g$(或$\mathbf{g}$)非零的**非齐次情形**，方程(18.0.3)和(18.0.4)除了当$\lambda$(或$\sigma$)为特征值外均可解——因为当$\lambda$(或$\sigma$)为特征值时积分算子(或矩阵)是奇异的。积分方程中的这种二分法称为**弗雷德霍姆择一性**。

第一类(Fredholm)方程通常是极其病态的。核作用到一个函数上通常起着光滑化作用,因而需要对该算子求逆的解,就会对输入的小变化或误差变得极其敏感。光滑的结果实际上常常丢失信息,而逆运算却无法找回它。这种方程已有了专门的处理方法,通常称为**反演问题**。一般说来,它必须利用解性质的一些先验知识来增加信息。这些先验知识在一定程度上便用来恢复丢失的信息。我们将第18.4节中介绍这些方法。

第二类非齐次**弗雷德霍姆**方程的病态程度通常要低得多。方程(18.0.4)可改写为

$$\int_a^b [K(t,s)-\sigma\delta(t-s)]f(s)ds=-\sigma g(t) \tag{18.0.5}$$

其中$\delta(t-s)$是一个狄拉克(Dirac)δ函数(为明确起见,我们将$\lambda$变为相应的$\sigma$)。如果$\sigma$的值足够大,则方程(18.0.5)实际上是对角占优的,因而是良态的。只有当$\sigma$很小时,我们才又回到病态情形。

第二类齐次**弗雷德霍姆**方程同样没有特别的病态。如果$K$是一个光滑算子,它便会将许多$f$值映为零,或接近零。因而,在$\sigma=0(\lambda\to\infty)$附近会出现大量退化或近似退化的特征值,但这对计算并未产生特别的困难,现在,我们可以看出,实际上使非齐次方程(18.0.5)不发生病态的$\sigma$值的大小,通常比对角占优情形所要求的$\sigma$值小得多。由于$\sigma$项会使所有特征值发生偏移,因此它要足够大,以使一个光滑算子的一系列零附近的特征值偏离零,那么合成的算子就变得可逆的(当然,除离散特征值外)。

**沃尔泰拉**(Volterra)**方程**是弗雷德霍姆方程的一种特殊情形,即当$s>t$时,核$K(t,s)=0$。扔掉不必要的积分部分,沃尔泰拉方程可写为积分上限为独立变量t的形式。**第一类沃尔泰拉方程**

$$g(t)=\int_a^t K(t,s)f(s)ds \tag{18.0.6}$$

和它的对应方程类似,也有矩阵方程(现在用分量写出)

$$\sum_{j=1}^{k} K_{kj}f_j=g_k \tag{18.0.7}$$

与方程(18.0.2)相比,我们发现沃尔泰拉方程对应着一个下(即左)三角矩阵$\mathbf{K}$,零项均在对角线上方。由第二章我们知道,这样的矩阵方程只需简单地通过前向代换就可解出。因此,沃尔泰拉方程的解法也同样直接了当。当实验测量的噪声不占绝对优势时,第一类沃尔泰拉方程基本上是非病态的。对积分取上限会产生一个极强的跃跃,它通常能破坏核所有的光滑性质。

**第二类沃尔泰拉方程**写为

$$f(t)=\int_a^t K(t,s)f(s)ds+g(t) \tag{18.0.8}$$

其对应的矩阵方程是

$$(\mathbf{K}-\mathbf{1})\cdot\mathbf{f}=\mathbf{g} \tag{18.0.9}$$

其中$\mathbf{K}$为下三角型矩阵。这些方程中没有$\lambda$的原因是:(i)在非齐次情形下($g$非零),$\lambda$可以被K吸收;(ii)而在齐次情形下($g=0$)下,有一定理,即核有界的第二类沃尔泰拉方程没有具有平方可积的特征函数的特征根。

我们将定义都限定在线性积分方程的情形。方程(18.0.1)或(18.0.6)在非线性情形下,被积函数将是$K(t,s,f(s))$而非$K(t,s)f(s)$;方程(18.0.4)或(18.0.8)在非线性情形下,被

积函数将是 $K(t,s,f(t),f(s))$。非线性弗雷德霍姆方程比相应的线性方程要复杂得多。幸运的是，在实际应用中这类方程出现得不多，本章我们基本上不考虑它们。但是求解非线性沃尔泰拉方程，通常只需对线性方程的算法稍加修改即可，这一点以后我们将会看到。

几乎所有积分方程的数值解法都要利用**求积分法则**，通常用**高斯**积分。读者现在可以复习一下第4.5节，特别是那节末尾较深的一些内容。

在下面各节，我们先讨论具有光滑核的第二类**弗雷德霍姆**积分方程（第18.1节）。该节也要讨论非平凡求积法则，但我们将只处理方程的良性系统。然后，回到沃尔泰拉方程（第18.2节），我们会发现，简单直接的方法通常很适合这些方程。

在第18.3节我们讨论如何处理奇异核情形，主要针对**弗雷德霍姆**方程（包括第一类和第二类）。奇异性要求有特殊的积分法则，但由于它们会破坏核的光滑性并使问题良态，因而我们有时也因祸得福。

在第18.4～18.7节，我们将讨论反演问题。第18.4节是对这个大专题的一个简介。

这里我们应该注意，第13.10节中讨论过的小波变换不仅适用于数据压缩和信号处理，而且了可以将一些积分方程变换成能快速求解的稀疏线性问题。作为阅读本章的一部分，读者也许会希望复习一下第13.10节。

我们必须坦率指出，有些课题，比如**积分-微分方程**，已超出本书的范围。如果想了解积分-微分方程的一些方法，见 Brunner[4]。

无庸置疑，仅用一章的篇幅只能论及积分方程这个复杂课题的一些最基本的方法。

参考文献和进一步读物：


Delves, L. M., and Mohamed, J. L. 1985, *Computational Methods for Integral Equations* (Cambridge, U.K.: Cambridge University Press). [1]

Linz, P. 1985, *Analytical and Numerical Methods for Volterra Equations* (Philadelphia: S.I.A.M.). [2]

Atkinson, K. E. 1976, *A Survey of Numerical Methods for the Solution of Fredholm Integral Equations of the Second Kind* (Philadelphia: S.I.A.M.) [3]

Brunner, H. 1988, in *Numerical Analysis* 1987, Pitman Research Notes in Mathematics Vol. 170, D.F. Griffiths and G.A. Watson, eds. (Harlow, Essex, U.K.: Longman Scientific and Technical), pp. 18～38. [4]


## 18.1 第二类弗雷德霍姆方程

我们想求解方程

$$f(t) - \lambda\int_a^b K(t,s)f(s)ds = g(t) \qquad (18.1.1)$$

中 $f(t)$ 的数值解。使用的方法称为**奈斯特姆**(Nystrom)**法**，这是一个非常基本的方法。它需要选取一些近似的**积分法则**：

$$\int_a^b y(s)ds = \sum_{j=1}^{N} w_j y(s_j) \qquad (18.1.2)$$

其中集 $\{w_j\}$ 是积分的法则的权，而 N 点 $\{s_j\}$ 集是横坐标。

我们应该采用什么积分法则呢？用低阶积分法则，比如重复梯形或辛卜生法则当然能解

积分方程。

但我们将看到，解法涉及 $O(N^3)$ 阶运算，因而最有效的方法倾向于用高阶积分法则以保持 $N$ 尽可能小。对于光滑、非奇异问题，没有什么方法能比得上高斯积分（如，高斯-勒让德积分，见第4.5节）。（对于非光滑或有奇异核，见第18.5节。）

Delves 和 Mohamed[1] 提出了比奈斯特姆方法更复杂的方法。对于简单的第二类弗雷德霍姆积分方程，他们认为："…这场比赛的当然胜利者是奈斯特姆（Nystrom）程序…采用 N 点高斯-勒让德法。该程序极其简单…这样的结果足以使一个数值分析学者深感叹服。"

如果将式(18.1.2)的积分法则应用到方程(18.1.1)上，我们得到

$$f(t) = \lambda\sum_{j=1}^{N} w_j K(t,s_j) f(s_j) + g(t) \qquad (18.1.3)$$

在积分点估算方程(18.1.3)：

$$f(t_i) = \lambda\sum_{j=1}^{N} w_j K(t_i,s_j) f(s_j) + g(t_i) \qquad (18.1.4)$$

令 $f_i$ 为向量 $f(t_i)$，$g_i$ 为向量 $g(t_i)$，$K_{ij}$ 为矩阵 $K(t_i,s_j)$，并定义

$$\tilde{K}_{ij} = K_{ij} w_j \qquad (18.1.5)$$

则方程(18.1.4)用矩阵写法，表示为

$$(\mathbf{1} - \lambda\tilde{\mathbf{K}}) \cdot \mathbf{f} = \mathbf{g} \qquad (18.1.6)$$

这是 $N$ 个未知量的 $N$ 个线性代数方程组，可以通过标准三角形分解法（第2.3节）来求解——正是在此出现了 $O(N^3)$ 次运算。解通常是良态的，除非 $\lambda$ 和一个特征值很接近。

求出积分点 $\{t_i\}$ 的解后，其它点 $t$ 的解怎样呢？简单的多项式插值不行的。这会破坏费尽全力才获得的精度。奈斯特姆法的关键点在于，采用方程(18.1.3)做插值公式以保持解的精度。

这里我们给出两个求解第二类线性弗雷德霍姆方程的程序。程序 **fred2** 先建立方程(18.1.6)，然后调用 **ludcmp** 和 **lubksb** 程序，利用 $LU$ 分解来求解。求高斯-勒让德积分是先通过调用 **gauleg** 得到权和横坐标。程序 **fred2** 要求读者提供一个返回 $g(t)$ 的外部函数及另一个返回 $\lambda K_{ij}$ 的外部函数。然后它返回积分点的解 $f$。它也返回积分点和权重。第二个程序 **fredin** 用这些数据来实现方程(18.1.3)的 Nystrom 的插值，并返回在区间 $[a,b]$ 内任意点的 $f$ 值。

```
#include "nrutil.h"

void fred2(int n, float a, float b, float t[], float f[], float w[],
    float (*g)(float), float (*ak)(float, float))
    解第二类线性弗雷德霍姆方程。输入 a 和 b 是积分限，n 是高斯积分点数，g 和 ak 是有用户提供的分别返回 g(t) 和
    λK(t,s)的外部函数。程序返回包含着高斯积分横坐标 t 及这些横坐标上的解 f 的数组 t[1..n]及 f[1..n]，同时已
    返回高斯权重数组 w[1..n]。与奈斯特姆插值程序 fredin 一起使用。
{
    void gauleg(float x1, float x2, float x[], float w[], int n);
    void lubksb(float **a, int n, int *indx, float b[]);
    void ludcmp(float **a, int n, int *indx, float *d);
    int i,j,*indx;
    float d,**omk;

    indx=ivector(1,n);
    omk=matrix(1,n,1,n);
```

```
    gauleg(a,b,t,w,n);                                         如果不用高斯-勒让德积分，
    for (i=1;i<=n;i++) {                                       用另一个程序替换 gauleg
        for (j=1;j<=n;j++) {                                   形成1−λK
            omk[i][j]=(float)(i == j)-(*ak)(t[i],t[j])*w[j];
        f[i]=(*g)(t[i]);
    }
    ludcmp(omk,n,indx,&d);                                     解线性方程
    lubksb(omk,n,indx,f);
    free_matrix(omk,1,n,1,n);
    free_ivector(indx,1,n);
}

float fredin(float x, int n, float a, float b, float t[], float f[],
    float w[], float (*g)(float), float (*ak)(float, float))
    给定包含着高斯积分横坐标和权重的数组 t[1..n]和 w[1..n]，由 fred2给出解数组 f[1..n]，该函数利用 Nystrom
    插值公式返回 x 点的 f 值。作为输入，a 和 b 是积分限，n 是高斯积分点数，g 和 ak 是用户提供的分别返回 g(t) 和 λK
    (t,s)的外部函数。
{
    int i;
    float sum=0.0;

    for (i=1;i<=n;i++) sum += (*ak)(x,t[i])*w[i]*f[i];
    return (*g)(x)+sum;
}
```

利用高斯积分法的一个缺点是，没有简单的办法来取得结果中误差的估计。最实用的方法是，将 $N$ 增加，例如 50%，将两次估计值之间的差别作为，用较大的 $N$ 值取得的结果中误差的一个保守估计。

现在，再看齐次方程的解。如果我们令 $\lambda=1/\sigma, g=0$，则方程(18.1.6)变为一个标准特征方程

$$\widetilde{\mathbf{K}}\cdot\mathbf{f}=\sigma\,\mathbf{f} \tag{18.1.7}$$

我们于是可以利用任何合适的矩阵特征程序来求解(见第十一章)。注意，如果我们的原始的问题有一个对称的核，那么矩阵 $\mathbf{K}$ 是对称的。但是，由于大部分积分法的权 $w_j$ 不相同，矩阵 $\widetilde{\mathbf{K}}$(方程18.1.5)并不对称。对称矩阵特征值问题要简单得多，因此我们应尽可能地保持对称性。假定权为正值(高斯积分就是如此)，我们可以定义对角矩阵 $\mathbf{D}=\mathrm{diag}(w_j)$ 及其平方根，$\mathbf{D}^{1/2}=\mathrm{diag}(\sqrt{w_j})$。方程(18.1.7)则变为

$$\mathbf{K}\cdot\mathbf{D}\cdot\mathbf{f}=\sigma\,\mathbf{f}$$

乘以 $\mathbf{D}^{1/2}$，我们得到

$$(\mathbf{D}^{1/2}\cdot\mathbf{K}\cdot\mathbf{D}^{1/2})\cdot\mathbf{h}=\sigma\,\mathbf{h} \tag{18.1.8}$$

其中，$\mathbf{h}=\mathbf{D}^{1/2}\cdot\mathbf{f}$。现在方程(18.1.8)便具有了对称特征值问题的形式。

方程(18.1.7)或(18.1.8)的解一般给出 $N$ 个特征值，其中 $N$ 是所用的积分点数。对于平方可积核，这些值将给出积分方程最小值 $N$ 个特征值的良好近似值。**有限秩的核**(也称为**退化或可分核**)仅有有限个非零特征值(也可能没有)。读者可以通过一组零特征值来判定这种情形以取得高精度。当增加 N 来提高它们的精度时，非零特征值的个数保持常值不变。这里需要注意：一个非退化的核可以有无穷多个以 $\sigma=0$ 为聚点的特征值。随着 $N$ 的增大，读者可以根据解的性质来区分这两种情形。如果是退化核，通常可用解析法求解，所有教材中都有这部分内容。

参考文献和进一步读物：

Delves, L. M., and Mohamed, J. L., 1985, *Computational Methods for Integral Equations* (Cambridge, U. K.: Cambridge University Press). [1]

## 18.2 沃尔泰拉方程

现在让我们再回到沃尔泰拉(Volterra)方程，其原型是第二类沃尔泰拉方程：

$$f(t)=\int_a^t K(t,s)f(s)ds+g(t) \tag{18.2.1}$$

沃尔泰拉方程的大部分算法从 $t=a$ 开始，随着不断地迭代逐渐求出解。在该意义上，它们不仅类似向前代换(见第18.0节)，而且类似于普通微分方程的初值问题。事实上，许多普通微分方程的算法都有沃尔泰拉方程的对应算法。

最简单的方法是，在均匀步长的网格点上求解方程：

$$t_i=a+ih,\quad i=0,1,\dots,N,\quad h\equiv\frac{b-a}{N} \tag{18.2.2}$$

为此必须选取一个积分法则。对于均匀网格，最简单的方法是梯形法则，方程(4.1.11)：

$$\int_a^{t_i}K(t_i,s)f(s)ds=h\left(\frac{1}{2}K_{i0}f_0+\sum_{j=1}^{i-1}K_{ij}f_j+\frac{1}{2}K_{ii}f_i\right) \tag{18.2.3}$$

因而方程(18.2.1)的梯形法是

$$\begin{aligned}&f_0=g_0\\&\left(1-\frac{1}{2}hK_{ii}\right)f_i=h\left(\frac{1}{2}K_{i0}f_0+\sum_{j=1}^{i-1}K_{ij}f_j\right)+g_i,\quad i=1,\dots,N\end{aligned} \tag{18.2.4}$$

(对于第一类沃尔泰拉方程，左式前面的1将不出现，$g$ 的符号相反，余下的讨论也有相应简单的变化。)

方程(18.2.4)是一个通过 $O(N^2)$次运算给出解的一个显式的方法。与弗雷德霍姆方程不同，它不需要解一个线性方程组。因此，沃尔泰拉方程常常比相应的弗雷德霍姆方程简单，我们已经知道，后者确实涉及到线性系统的反演，有时甚至是很大的线性系统的反演。

求解沃尔泰拉方程的高效率在一定程度上被下述事实所抵消，即在实际应用中，该方程以方程组形式出现更为常见。如果将方程(18.2.1)改写成以 $m$ 个函数 $f(t)$ 为向量的方程，则核 $k(t,s)$ 为一个 $m\times m$ 矩阵。方程(18.2.4)也应看成为一个向量方程。对每个 $i$，我们必须利用高斯消元法求解 $m\times m$ 线性代数方程组。

下面的 **voltra** 方程实现了该算法。要求读者提供一个返回在点 $t$ 向量 $g(t)$ 的第 $K$ 个函数的外部函数；以及一个返回矩阵 $K(t,s)$ 在$(t,s)$处的$(k,l)$元素的外部函数。然后，**voltra** 程序返回规则步长点 $t_i$ 的向量 $f(t)$。

```
#include "nrutil.h"

void voltra(int n, int m, float t0, float h, float *t, float **f,
    float (*g)(int, float), float (*ak)(int, int, float, float))
    用扩展梯形法求解 m 个第二类线性沃尔泰拉方程组。作为输入，t0是积分的起始点，n-1是长为 h 的步数。g(k,t)是
    用户提供的返回 gk(t)的外部函数，而 ak(k,l,t,s)则是一个由用户提供的返回矩阵 K(t,s)的(k,l)元素的外部函
```

数。该积分的解返回到 f[1..m][1..n]，相应的横坐标为 t[1..n]。

```
{
    void lubksb(float **a, int n, int *indx, float b[]);
    void ludcmp(float **a, int n, int *indx, float *d);
    int i,j,k,l,*indx;
    float d,sum,**a,*b;

    indx=ivector(1,m);
    a=matrix(1,m,1,m);
    b=vector(1,m);
    t[1]=t0;
    for (k=1;k<=m;k++) f[k][1]=(*g)(k,t[1]);        初始赋值
    for (i=2;i<=n;i++) {                             取步长 h
        t[i]=t[i-1]+h;
        for (k=1;k<=m;k++) {
            sum=(*g)(k,t[i]);                        将线性方程的右端项
            for (l=1;l<=m;l++) {                       累加到sum中
                sum += 0.5*h*(*ak)(k,l,t[i],t[1])*f[l][1];
                for (j=2;j<i;j++)
                    sum += h*(*ak)(k,l,t[i],t[j])*f[l][j];
                a[k][l]=(k == l)-0.5*h*(*ak)(k,l,t[i],t[i]);    左端项放到矩阵 a 中
            }
            b[k]=sum;
        }
        ludcmp(a,m,indx,&d);                         解线性方程
        lubksb(a,m,indx,b);
        for (k=1;k<=m;k++) f[k][i]=b[k];
    }
    free_vector(b,1,m);
    free_matrix(a,1,m,1,m);
    free_ivector(indx,1,m);
}
```

对于非线性沃尔泰拉方程，方程(18.2.4)仍然成立，但乘积 $K_{ii}f_i$ 要变为 $K_{ii}(f_i)$，另两个乘积 $K$ 和 $f$ 也做类似变化。因而对每个 $i$，我们都已知右端项来求非线性方程的解 $f_i$。如果步长不太大的话，则采用具有 $f_{i-1}$ 的初始猜测值的牛顿法(第9.4节或第9.6节)，其效果通常很好。

我们认为，用高阶方法解沃尔泰拉方程，不如用高阶方法解弗雷德霍姆方程那样重要，因为沃尔泰拉方程相对易解。不过和这个课题有关的文献却很多，随之也产生了一些困难。首先，当初始几点的值未知时，任何通过同时在几个积分点运算来获取高阶的方法，都需要采用一个特别的方法来开始。

其次，稳定积分法则会使积分方程产生意外的不稳定性。例如，假定我们想用辛卜生法则来代替上述算法中的梯形法。辛卜生法自然地对长为 $2h$ 的间隔作积分，因此我们很容易得到偶网格点上的函数值。对于奇网格点，我们可以附加用一次梯形法。但应该加在积分的哪一端呢？我们可以先用梯形法先做一步，再用辛卜生法；也可以先用辛卜生法，最后一步再用梯形法。奇怪的是，前者不稳定，而后者稳定。

和上文给出的梯形法可以一起使用的一个简单方法是理查得森外推法：用步长 $h$ 和 $h/2$ 求解。然后，假定误差的量为 $h^2$，计算

$$f_E = \frac{4f(h/2) - f(h)}{3} \tag{18.2.5}$$

这个方法也适用于龙贝格积分。

目前普遍认为，高阶方法中最好的是块-块法(见[1])。另一个重要的课题是，变步长法

的应用，这在 $K$ 或 $f$ 有尖峰变化时更有效。变步长法比微分方程中相应的方法要复杂的多，建议读者参考文献[1,2]做进一步的讨论。

读者也应该注意被积函数的奇异性。如果有奇异性，那么参考第18.3节可以找到一些新方法。

参考文献和进一步读物：


Linz, P. 1985, *Analytical and Numerical Methods for Volterra Equations* (Philadelphia: S. I. A. M.). [1]

Delves, L. M., and Mohamed, J. L. 1985, *Computational Methods for Integral Equations* (Cambridge, U. K.: Cambrige University Press). [2]


## 18.3 具有奇异核的积分方程

许多积分方程，或核或解或核与解都有奇异性。如果忽略这些奇异性，那么简单的积分法对 $N$ 的收敛性就很差。如何最好地处理奇异性，有时需要一定的技巧。

我们从一些简单的建议开始：

1. 可积奇异点常常可以通过变量替换消除掉。例如在 $s=0$ 附近 $K(t,s)$ 的 $s^{1/2}$ 或 $s^{-1/2}$ 奇异性可通过变换 $z=s^{1/2}$ 而消除掉。注意，我们假定奇异性限制在 $K$ 上，而积分实际上涉及的是乘积 $K(t,s)\cdot f(s)$，因此需要"固定"的正是这个乘积。理想情形下，必须证明乘积的奇异性后，才能进行数值求解和采用适当的方法。然而，奇异核常常并不产生奇异解 $f(t)$。(例如，高奇异性核 $K(t,s)=\delta(t-s)$ 实际上是一个恒等算子。)

2. 如果 $K(t,s)$ 能分解为 $w(s)\bar{K}(t,s)$，其中 $w(s)$ 是奇异的，而 $\bar{K}(t,s)$ 是光滑的，那么以 $w(s)$ 为权函数的高斯积分法效果很好。甚至当分解近似成立时，收敛性常常也会改进很多。需要做的，只是将 **fred2** 程序中的 **gauleg** 用另一个积分程序来代替即可。第4.5节介绍了如何构造这样的积分；或者也可以在标准参考文献[1,2]中查找列成表的横坐标和权。当然，$K$ 必须用 $\bar{K}$ 来替换。

该方法是**乘积奈斯特姆**(Ngstrom)**法**的一种特殊情形，乘积奈斯特姆法从 $K$ 中分解出一个同时依赖于 $t$ 和 $s$ 的奇异项 $p(t,s)$，并为其高斯积分构造了合适的权。一般情形的计算相当复杂，因为权不仅依赖于 $t_i$ 的选取，而且依赖于 $p(t,s)$ 的形式。

我们较喜欢用均匀网格点来实现乘积奈斯特姆法，采用把对任意权函数适用的扩展辛卜生3/8法(方程(4.1.5))进行推广后得到的积分方案。我们在下一节讨论该方法。

3. 当核不严格但却"几乎"奇异时，一些特殊的积分公式就很有用。例如，当核集中在 $t=s$ 附近，其变化范围远远小于解 $f(t)$ 的变化范围。在这种情形下，积分公式可以通过计算核 $K(t,s)$ 在表格点 $t_i$ 的最初几个矩，并利用多项式或样条对 $f(s)$ 的局部近似来获取。在这种方案下，核的窄宽度变得有用了，而不成为负担：当核的宽度为零时，积分变为精确值。

4. 无限区域的积分也是奇异性的一种形式。在很大的有限值处将区域截断应该是不得已的作法。如果核很快地趋于零，那么高斯-勒让德[$w\sim\exp(-\alpha s)$]或高斯-埃尔米特[$w\sim\exp(-s^2)$]积分的效果应该很好。长尾函数通常采用变换

$$s=\frac{2\alpha}{z+1}-\alpha \tag{18.3.1}$$

它将 $0<s<\infty$ 映到 $1>z>-1$，这样就可以用高斯-勒让德积分。其中 $\alpha>0$ 为一可调整的常数，用来改进收敛性。

5. 实际应用中 $K(t,s)$ 常常是沿对角线 $t=s$ 有奇异性。这时奈斯特姆法完全不适用了，因为核在 $(t,t)$ 处得到估计值。而**奇异性减法**可能是一个补救的方法

$$\int_a^b K(t,s)f(s)ds=\int_a^b K(t,s)[f(s)-f(t)]ds+\int_a^b K(t,s)f(t)ds$$

$$= \int_a^b K(t,s)[f(s) - f(t)]ds + r(t)f(t) \tag{18.3.2}$$

其中 $r(t) = \int_a^b K(t,s)$，可以解析或数值计算求得。现在如果右端第一项是奇异的，那么，我们总可以用奈斯特姆法。不同于方程(18.1.4)，我们得到

$$f_i - \lambda \sum_{\substack{j=1 \\ j \neq i}}^{N} w_j K_{ij}[f_j - f_i] - \lambda r_i f_i = g_i \tag{18.3.3}$$

有时，在核完全正规化之前减法过程要重复多次，详见[3]。(我们认为，阅读一下有不同的方式来处理其他奇异性的文献也许会更好些。)

## 18.3.1 具有任意权的均匀网格上的积分

一般说来，可以找到 $n$ 点线性积分法则，它能使函数 $f(x)$ 乘以任意权函数 $w(x)$，对任意积分区间 $(a,b)$ 上的积分，近似为权乘以函数 $f(x)$ 的几个均匀分划值的和，比如取 $x=kh,(k+1)h,\cdots,(k+n-1)h$。推导这种积分法则的一般方案是，写出该积分法精确成立时，对函数 $f(x)=$ 常数，$x,x^2\cdots x^{n-1}$ 必须满足的 $n$ 个线性方程，然后解这些方程求出系数。假定已知权函数在同一积分区域的矩

$$W_n \equiv \frac{1}{h^n}\int_a^b x^n w(x)dx \tag{18.3.4}$$

那么，我们可以解析地使问题得到彻底的解决。若(通常也会这样)$b-a$ 和 $h$ 在比例，则约因子 $h^{-n}$ 用来使 $W_n$ 和 $h$ 的尺度相同。

在四点情况下，实现该想法可给出结果

$$\begin{aligned}
&\int_a^b w(x)f(x)dx = \\
&\frac{1}{6}f(kh)[(k-1)(k-2)(k-3)W_0 - (3k^2-12k+11)W_1 + 3(k-2)W_2 - W_3] \\
&+ \frac{1}{2}f([k+1]h)[-k(k-2)(k+3)W_0 - (3k^2-10k+6)W_1 - (3k-5)W_2 + W_3] \\
&+ \frac{1}{2}f([k+2]h)[k(k+1)(k+3)W_0 - (3k^2+8k-3)W_1 + (3k-4)W_2 - W_3] \\
&+ \frac{1}{6}f([k+3]h)[-k(k-1)(k+2)W_0 + (3k^2+6k+2)W_1 - 3(k+1)W_2 + W_3]
\end{aligned} \tag{18.3.5}$$

虽然括号中的项表面上和 $k^3$ 的尺度一致，但实际上 $O(k^3)$ 和 $O(k)$ 均被消去了。

$(a,b)$ 的各种不同的选法，使得方程(18.3.5)有多种特殊化形式。较明显的选法是 $a=kh,b=(k+3)h$。在此情形下，我们得到一个由辛卜生 3/8 法则(方程(4.1.5))推广产生的四点积分法。事实上，我们可以通过置 $w(x)=1$ 来恢复这个特殊情形，此时式(18.3.4)变成

$$W_n = \frac{h}{n+1}[(k+3)^{n+1} - k^{n+1}] \tag{18.3.6}$$

方程(18.3.5)中方括号里四项的每项都变得和 $k$ 无关，方程(18.3.5)实际上变为

$$\int_{kh}^{(k+3)h} f(x)dx = \frac{3h}{8}f(kh) + \frac{9h}{8}f([k+1]h) + \frac{9h}{8}f([k+2]h) + \frac{3h}{8}f([k+3]h) \tag{18.3.7}$$

再回到一般的 $w(x)$ 的情形，$a$ 和 $b$ 的另外一些选法也很有用。例如，我们可能想将 $(a,b)$ 选作为 $([k+1]h,[k+3]h)$ 或 $([k+2]h,[k+3]h)$。只要我们构造一个其积分区间数不是 3 的倍数的扩展法，那么精度就不会有影响；积分将用四个值 $f(kh),\cdots,f([k+3]h)$ 来估计。更有用的是将 $(a,b)$ 选取为 $([k+1]h,[k+2]h)$，这样就可用四点法来求集中在一个单一区间上的积分。当这些权被纳入一个扩展的公式时，它们给出了具有光滑系数的积分格式，即没有象辛卜生法中那样进行 2、4、2、4、2 的交换。(事实上，这是我们推导方程(4.1.14)时曾经用过的技巧，也许现在想复习一下这些内容。)

所有这些法则都和扩展辛卜生法同阶，即 $f(x)$ 为精确的三次多项式。如果想用低阶法则，也可以类似地求得。三点公式是

$$
\begin{aligned}
\int_{kh}^{(k+2)h} w(x)f(x)dx =& \frac{1}{2}f(kh)\left[(k+1)(k+2)W_0 - (2k+3)W_1 + W_2\right] \\
&+ f([k+1]h)\left[-k(k+2)W_0 + 2(k+1)W_1 - W_2\right] \\
&+ \frac{1}{2}f([k+2]h)\left[k(k+1)W_0 - (2k+1)W_1 + W_2\right]
\end{aligned}
\tag{18.3.8}
$$

这里简单特殊的情形是取 $w(x)=1$，则

$$
W_n = \frac{h}{n+1}\left[(k+2)^{n+1} - k^{n+1}\right] \tag{18.3.9}
$$

方程(18.3.8)变成了辛卜生法：

$$
\int_{kh}^{(k+2)h} f(x)dx = \frac{h}{3}f(kh) + \frac{4h}{3}f([k+1]h) + \frac{h}{3}f([k+2]h) \tag{18.3.10}
$$

但是对于非常数的权函数 $w(x)$，方程(18.3.8)却给出了一个比辛卜生低一阶的法则，因为常数值情形具有的特别的对称现象，会给它们带来很大的方便。

两点公式可简单地写为

$$
\int_{kh}^{(k+1)h} w(x)f(x)dx = f(kh)\left[(k+1)W_0 - W_1\right] + f([k+1]h)\left[-kW_0 + W_1\right] \tag{18.3.11}
$$

下面的程序 **wwghts** 是利用上述公式得到的，它返回在积分区间 $(a,b)=(0,[N-1]h)$ 上的一个扩展的 $N$ 点积分法。输入到 **wwghts** 中的是一个由用户提供的程序 **Kermom**，调用它可求得 $w(x)$ 的矩阵的最初四个不定积分矩，即

$$
F_m(y) \equiv \int^y s^m w(s)ds \qquad m = 0,1,2,3 \tag{18.3.12}
$$

(下限任意，可以根据需要选取。)千万注意，当 $N<4$ 时，调用 **wwghts** 会返回一个比辛卜生法低阶的法则，读者应该适当地组织问题以避免该情形的出现。

```
void wwghts(float wghts[], int n, float h, void (*kermom)(double [], double, int))
    wghts[1..n]中构造了函数 f(x)从0到(n-1)h 的 n 点等区间积分的权重与任意一个(可能奇异)权函数 w(x)的乘
    积，而 w(x)的不定积分矩 Fn(y)从用户提供的 Kermom 程序中获取。
{
    int j,k;
    double wold[5],wnew[5],w[5],hh,hi,c,fac,a,b;
    即使交界处为单精度，内部计算也采用双精度

    hh=h;
    hi=1.0/hh;
    for (j=1;j<=n;j++) wghts[j]=0.0;  零化权重以便累加它们

    (*kermom)(wold,0.0,4);                  在低端估计不定积分
    if (n >= 4) {                           用尽可能高的阶
        b=0.0;                              对另一个问题，可能要改变该下限
        for (j=1;j<=n-3;j++) {
            c=j-1;                          在方程(18.3.5)中这称为 k
            a=b;                            对该步置上限和下限
            b=a+hh;
            if (j == n-3) b=(n-1)*hh;       最后一个区间：到结尾完全一致
            (*kermom)(wnew,b,4);
            for (fac=1.0,k=1;k<=4;k++,fac*=hi)      方程(18.3.4)
                w[k]=(wnew[k]-wold[k])*fac;
            wghts[j] += (                           方程(18.3.5)
                ((c+1.0)*(c+2.0)*(c+3.0)*w[1]
                -(11.0+c*(12.0+c*3.0))*w[2]
                +3.0*(c+2.0)*w[3]-w[4])/6.0);
            wghts[j+1] += (
                (-c*(c+2.0)*(c+3.0)*w[1]
                +(6.0+c*(10.0+c*3.0))*w[2]
                -(3.0*c+5.0)*w[3]+w[4])*0.5);
```

```
            wghts[j+2] += (
                (c*(c+1.0)*(c+3.0)*w[1]
                -(3.0+c*(8.0+c*3.0))*w[2]
                +(3.0*c+4.0)*w[3]-w[4])*0.5);
            wghts[j+3] += (
                (-c*(c+1.0)*(c+2.0)*w[1]
                +(2.0+c*(6.0+c*3.0))*w[2]
                -3.0*(c+1.0)*w[3]+w[4])/6.0);
            for (k=1;k<=4;k++) wold[k]=wnew[k];        重置矩的下限
        }
    } else if (n == 3) {                                低阶情形，不推荐
        (*kermom)(wnew,hh+hh,3);
        w[1]=wnew[1]-wold[1];
        w[2]=hi*(wnew[2]-wold[2]);
        w[3]=hi*hi*(wnew[3]-wold[3]);
        wghts[1]=w[1]-1.5*w[2]+0.5*w[3];
        wghts[2]=2.0*w[2]-w[3];
        wghts[3]=0.5*(w[3]-w[2]);
    } else if (n == 2) {
        (*kermom)(wnew,hh,2);
        wghts[1]=wnew[1]-wold[1]-(wghts[2]=hi*(wnew[2]-wold[2]));
    }
}
```

现在我们给出一个如何应用 wwghts 来解奇异积分方程的例子。

## 18.3.2 实例：对角奇异核

作为一个特殊例子，考虑积分方程

$$f(x)-\int_0^{\pi}K(x,y)f(y)dy=\sin x \tag{18.3.13}$$

它具有一个(任意选取)很难处理的核

$$K(x,y)=\cos x\cos y\times\begin{cases}\ln(x-y) & y<x\\ \sqrt{y-x} & y\geq x\end{cases} \tag{18.3.14}$$

对角线左侧出现对数奇异性，同时在右侧有平方根不连续。

第一步(该情形下用解析法)是对核的奇异部分求必需的积分矩，即方程(18.3.12)。由于在一个固定 $x$ 值上求这些积分，因此我们可选 $x$ 作为下限。对任意确定的 $y$ 值，则需要求的不定积分或是

$$F_m(y;x)=\int_x^y s^m(s-x)^{1/2}ds=\int_0^{y-x}(x+t)^m t^{1/2}dt \qquad 若\ y\geq x \tag{18.3.15}$$

或是

$$F_m(y;x)=\int_x^y s^m\ln(x-s)ds=\int_0^{x-y}(x-t)^m\ln t\,dt \qquad 若\ y<x \tag{18.3.16}$$

(其中第二个等式中的变量在每种情形下都有变化。)为了用解析法求这些积分(实际上，我们用了一条符号积分包!)，我们将这些合成公式都组装在下述程序中。注意 $w(j+1)$ 返回 $F_j(y;x)$。

```
#include <math.h>

extern double x;                     在 quadmx 中定义

void kermom(double w[], double y, int m)
    w[1..m]返回由核的奇异部分的一行中前 m 个不定积分矩。(本例中，m 具体取为 4。)输入变量 y 标记列，而全局变
    量 x 为行。我们可将 x 取为积分的下限。因而积分矩或者完全返回到对角线左侧，或者完全返回到对角线右侧。
{
    double d,df,clog,x2,x3,x4,y2;

    if (y >= x) {
        d=y-x;
        df=2.0*sqrt(d)*d;
```

```
        w[1]=df/3.0;
        w[2]=df*(x/3.0+d/5.0);
        w[3]=df*((x/3.0 + 0.4*d)*x + d*d/7.0);
        w[4]=df*(((x/3.0 + 0.6*d)*x + 3.0*d*d/7.0)*x+d*d*d/9.0);
    } else {
        x3=(x2=x*x)*x;
        x4=x2*x2;
        y2=y*y;
        d=x-y;
        w[1]=d*((clog=log(d))-1.0);
        w[2] = -0.25*(3.0*x+y-2.0*clog*(x+y))*d;
        w[3]=(-11.0*x3+y*(6.0*x2+y*(3.0*x+2.0*y))
            +6.0*clog*(x3-y*y2))/18.0;
        w[4]=(-25.0*x4+y*(12.0*x3+y*(6.0*x2+y*
            (4.0*x+3.0*y)))+12.0*clog*(x4-(y2*y2)))/48.0;
    }
}
```

第二步，我们写一个构造积分矩阵的程序。

```
#include <math.h>
#include "nrutil.h"
#define PI 3.14159265

double x;                              与 Kermom 传递的参数

void quadmx(float **a, int n)
    在a[1..n][1..n]中构造一个以第二类弗雷德霍姆积分方程为例的求积矩阵。该程序计算了核的非奇异部分，而
    积分核的奇异部分的积分权通过调用 wwghts 得到。提供核奇异部分的不定积分矩的外部程序 kermom 被传输
    到 wwghts 中。

    void kermom(double w[], double y, int m);
    void wwghts(float wghts[], int n, float h,
        void (*kermom)(double [], double, int));
    int j,k;
    float h,*wt,xx,cx;

    wt=vector(1,n);
    h=PI/(n-1);
    for (j=1;j<=n;j++) {
        x=xx=(j-1)*h;                                          使 x 为全局变量供 kermom 用
        wwghts(wt,n,h,kermom);
        cx=cos(xx);                                            非奇异核部分
        for (k=1;k<=n;k++) a[j][k]=wt[k]*cx*cos((k-1)*h);      把核的所有部分加起来
    ++a[j][j];                                                 由于第二类方程，和 h 无关的对角部分

    free_vector(wt,1,n);
}
```

最后一步，对任意右端项，此处为 sin x，求解线性方程组。

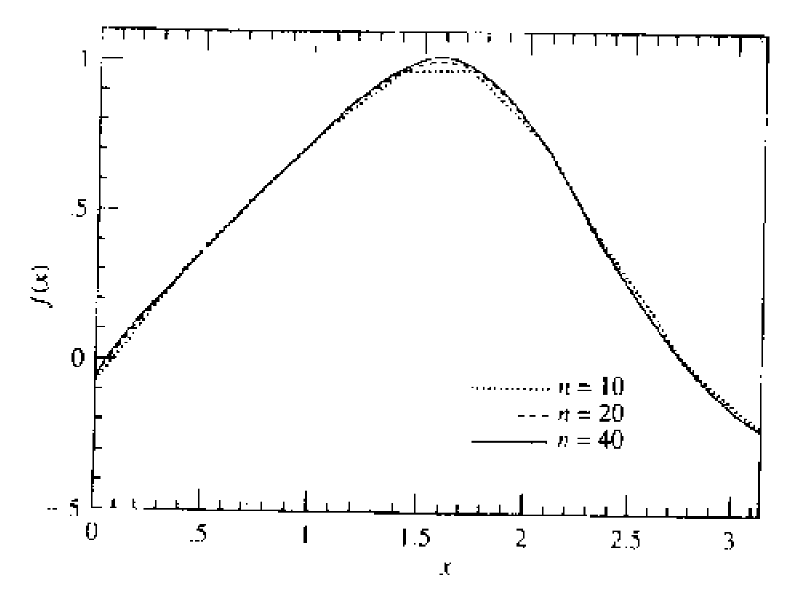
```
#include <stdio.h>
#include <math.h>
#include "nrutil.h"
#define PI 3.14159265                  这里网格的大小是确定的
#define N 40

int main(void)    /* 程序 fredex */
    该示例程序展示了怎样用乘积奈斯特罗姆法，和专门针对一个特定的奇异核，构造的一个积分法则，来求解
    第二类弗雷德霍姆方程。
```

```
void lubksb(float **a, int n, int *indx, float b[]);
void ludcmp(float **a, int n, int *indx, float *d);
void quadmx(float **a, int n);
float **a,d,*g,x;
int *indx,j;

indx=ivector(1,N);
a=matrix(1,N,1,N);
g=vector(1,N);
quadmx(a,N);                                           做积分矩阵;所有操作都在此
ludcmp(a,N,indx,&d);                                   分解矩阵
for (j=1;j<=N;j++) g[j]=sin((j-1)*PI/(N-1));           构造右端项,这里为sin x
lubksb(a,N,indx,g);                                    回代
for (j=1;j<=N;j++) {                                   写出解
    x=(j-1)*PI/(N-1);
    printf("%6.2d %12.6f %12.6f\n",j,x,g[j]);
}
free_vector(g,1,N);
free_matrix(a,1,N,1,N);
free_ivector(indx,1,N);
return 0;
```

当$N=40$时,该程序给出的精度约在$10^{-5}$量级。尽管是高奇异性核,精度增加的速度为$N^{-2}$(各个梯形积分方案也应如此)。图18.3.1显示了求出的解,同时也画出了N值较小的解,可以看出,这些解相当可信。应注意到,尽管核在通常情形下是奇异的,但解却仍是光滑的。




网格大小分别为$N=10$、20、40,解的离散值通过直线连起来。实际应用中,可以将小$N$值的解作内插使得曲线更光滑

图18.3.1 积分方程范例(18.3.14)的解

参考文献和进一步读物:


Abramowitz, M., and Stegun, I. A. 1964 *Handbook of Mathematical Functions*, Applied Mathematics Series, vol. 55. [1]

Stroud, A. H., and Secrest, D. 1966, *Gaussian Quadrature Formulas* (Englewood Cliffs, NJ: Prentice-Hall). [2]

Delves, L. M., and Mohamed, J. L. 1985, *Computational Methods for Integral Equations* (Cambridge, U. K.: Camridge University Press). [3]

Atkinson, K. E. 1976, *A Survey of Numerical Methods for the Solution of Fredholm Integral Equations of the Second Kind* (Philadelphia: S. I. A. M.). [1]

## 18.4 反演问题与先验信息的利用

应有一对数学上的基本性质会使得后面的讨论很方便。假定 u 是个"未知"向量,我们想通过一些最小化法则来确定它。令 $\mathscr{A}[\mathbf{u}]>0$ 及 $\mathscr{B}[\mathbf{u}]>0$ 为 u 的两个正值函数,则可以通过

$$\text{或使 } \mathscr{A}[\mathbf{u}] \text{ 最小} \quad \text{或} \quad \text{使 } \mathscr{B}[\mathbf{u}] \text{ 最小} \tag{18.4.1}$$

来设法确定 u。(自然,这些方法通常会确定出不同的 u 值。)另一种可能是,假定我们想先特别地限制 $\mathscr{B}[\mathbf{u}]$ 的取值,比如取 $b$,然后使 $\mathscr{A}[\mathbf{u}]$ 最小。拉格良日乘子法给出变分

$$\frac{\partial}{\partial \mathbf{u}}\{\mathscr{A}[\mathbf{u}]+\lambda_1(\mathscr{B}[\mathbf{u}]-b)\}=\frac{\partial}{\partial \mathbf{u}}(\mathscr{A}[\mathbf{u}]-\lambda_1\mathscr{B}[\mathbf{u}])=0 \tag{18.4.2}$$

其中 $\lambda_1$ 为拉格朗日乘子。注意第二个等式中没有 $b$,因为它和 u 无关。

接着,假定我们改变主意,限制 $\mathscr{A}[\mathbf{u}]$ 取特别值 $a$,使 $\mathscr{B}[\mathbf{u}]$ 最小。与方程(18.4.2)不同,我们得到

$$\frac{\partial}{\partial \mathbf{u}}\{\mathscr{B}[\mathbf{u}]+\lambda_2(\mathscr{A}[\mathbf{u}]-a)\}=\frac{\partial}{\partial \mathbf{u}}(\mathscr{B}[\mathbf{u}]+\lambda_2\mathscr{A}[\mathbf{u}])=0 \tag{18.4.3}$$

这里,$\lambda_2$ 为拉格朗日乘子。方程(18.4.3)乘以常数 $1/\lambda_2$,并令 $1/\lambda_2$ 等于 $\lambda_1$,我们发现这两种情形下的变分实际上是完全一样的。它们都满足同样的含一个参数的解系,如 $\mathbf{u}(\lambda_1)$。当 $\lambda_1$ 从 0 变到 $\infty$ 时,$\mathbf{u}(\lambda_1)$ 沿着一条所谓的**交替换位曲线**,在使 $\mathscr{A}$ 最小和使 $\mathscr{B}$ 最小之间变化。该曲线上的任何解都可以看作:或(i)限制 $\mathscr{B}$ 的取值,使 $\mathscr{A}$ 最小;或(ii)限制 $\mathscr{A}$ 的取值,使 $\mathscr{B}$ 最小;或(iii)使 $\mathscr{A}+\lambda_1\mathscr{B}$ 在权意义下最小。

第二个基本点与**退化**的最小化法则有关。在上述例子中,假定现在 $\mathscr{A}[\mathbf{u}]$ 特别地取为

$$\mathscr{A}[\mathbf{u}]=|\mathbf{A}\cdot\mathbf{u}-\mathbf{c}|^2 \tag{18.4.4}$$

其中 A 为矩阵,c 为向量。若 A 的行小于列,或 A 为方阵但却为退化矩阵(有一个非普通的零空间,见第2.6节,特别是图2.6.1),则使 $\mathscr{A}[\mathbf{u}]$ 最小也不能得到 u 的唯一解。(要知为什么,请复习第15.4节。注意,对于行小于列的一个"设计矩阵"A,正规方程(15.4.10)中的矩阵 $\mathbf{A}^T\cdot\mathbf{A}$ 是退化的)。但是,如果我们给 $\mathscr{A}[\mathbf{u}]$ 加上 $\lambda$ 与一个非退化二次型 $\mathscr{B}[\mathbf{u}]$ 的乘积,比如 $\mathbf{u}\cdot\mathbf{H}\cdot\mathbf{u}$,其中 H 为一确定的正值矩阵,则使 $\mathscr{A}[\mathbf{u}]+\lambda\mathscr{B}[\mathbf{u}]$ 最小,将得到 u 的唯一解,(两个二次型的和仍是一个二次型,第二部分保证非退化性)。

综合以上两个点,我们可以得到这样的结论。当一个二个次最小化法则加一个二次型限制,且两个都取正值时,只要其中之一非退化就能使整个问题的处于良态。现在我们可以着手解决反演问题了。

### 18.4.1 零阶正则化反演问题

假定 $u(x)$ 是一些未知的或基本的物理过程,(未知的和基本都由 $u$ 代表!)我们希望通过 $N$ 个测量值 $c_i, i=1,\cdots N$ 能确定 $u$。$u(x)$ 和 $c_i$ 之间的关系是,每个 $c_i$ 通过它自已的线性响应核 $r_i$ 来测定 $u(x)$ 在某一方面的性质,希望这些性质截然不同),其测量误差为 $n_i$。换言

之

$$c_i \equiv s_i - n_i = \int r_i(x)u(x)dx + n_i \tag{18.4.5}$$

（比较上式与方程13.3.1和13.3.2）在线性假设的范围内，这是个相当一般的公式。$c_i$ 可以近似地表示 $u(x)$ 在某些位置 $x_i$ 上的值，此时 $r_i(x)$ 则将具有集中在 $x=x_i$ 附近且窄或很窄的响应的形式。或者，$c_i$ 可以"住"在一个和 $u(x)$ 完全不同的函数空间里，例如测量 $u(x)$ 不同的傅里叶分量。

**反演问题**即是：给定 $c_i$ 和 $r_i$，也许还有误差 $n_i$ 的一些信息，例如它们的协方矩阵。

$$S_{ij} \equiv \text{Covar}[n_i, n_j] \tag{18.4.6}$$

我们应怎样做，才能找到一个 $u(x)$ 的最佳统计估算值，称为 $\hat{u}(x)$。

非常明显，这是个病态问题。但无论如何，我们如何能够根据有限个离散值 $c_i$ 来重建整个函数 $u(x)$ 呢？无论正式地还是非正式地，在科学研究中我们总在做这样的事。我们例行地测量"足够多的点"，然后"过它们画条线"，这样做时，我们总要作一些假定，或是关于原函数 $u(x)$ 的性质，或是关于响应函数 $r_i(x)$ 的性质，或是二者皆有。我们现在的目的是，将这些假定公式化，并将我们研究工作最大限度地扩展到，测量值和基本函数处于完全不同的函数空间的情况中去。（如怎样通过一些傅里叶系数的散布来"画条曲线"？）

我们并不真正想得到每个 $x$ 点的函数 $u(x)$。我们只想得到 $M$ 个离散点 $x_\mu$，$\mu=1,2\cdots M$，其中 $M$ 充分大，又 $x_\mu$ 充分均匀地分布，致使 $u(x)$ 或 $r_i(x)$ 在任意 $x_\mu$ 和 $x_{\mu+1}$ 之间的变化都不大。（此处及下文我们均用希腊字母，例如 $\mu$，来表示基本处理空间中的值，而用罗马字，比如 $i$ 来表示直接观察到的值。）对于 $x_\mu$ 这样稠密的一个集合，我们可用一个积分

$$c_i = \sum_\mu R_{i\mu}u(x_\mu) + n_i \tag{18.4.7}$$

来代替方程(18.4.5)，其中 N×M 矩阵 R 的分量为

$$R_{i\mu} = r_i(x_\mu)(x_{\mu+1} - x_{\mu-1})/2 \tag{18.4.8}$$

（或任何其它简单的积分　　用哪种积分并不重要）。实际应用中，我们认为方程(18.4.5)和方程(18.4.7)等价。

怎样从如方程(18.4.7)这样的方程组中解出未知的 $u(x_\mu)$ 呢？下面的方法并不好，但却包含着一些正确思想的萌芽：构造 $\chi^2$ 来衡量模型 $\hat{u}(x)$ 和测量数据的符合程度

$$\begin{aligned}\chi^2 &= \sum_{i=1}^{N}\sum_{j=1}^{N}\left[c_i - \sum_{\mu=1}^{M}R_{i\mu}\hat{u}(x_\mu)\right]S_{ij}^{-1}\left[c_j - \sum_{\mu=1}^{M}R_{j\mu}\hat{u}(x_\mu)\right] \\ &\approx \sum_{i=1}^{N}\left[\frac{c_i - \sum_{\mu=1}^{M}R_{i\mu}\hat{u}(x_\mu)}{\sigma_i}\right]^2\end{aligned} \tag{18.4.9}$$

（与方程(15.1.5)比较）。这里 $\mathbf{S}^{-1}$ 是协方差矩阵的逆，如果不在对角线上的协方差可以忽略，则约等号成立，$\sigma_i \equiv (\text{Covar}[i,i])^{1/2}$。

现在可以用第15.4节中奇异值分解法(SVD)来找使方程(18.4.9)最小的向量 $\mathbf{u}$，不要试图去用正规方程的方法，我们已经讨论过，由于 $M$ 比 $N$ 大，它们将是奇异的，因而 SVD 过程必定会找到很多零奇异值，表明解具有很高的不唯一性。在无穷多退化解中（大部分解对任意大的 $\hat{u}(x_\mu)$ 具有不好的性质），SVD 会选出一个在下述意义下的最小 $\hat{\mathbf{u}}$ 值

$$\sum_\mu [\hat{u}(x_\mu)]^2 \quad \text{为一个最小值} \tag{18.4.10}$$

（见图2.6.1）。这个解常称为**基本解**。它是所谓**零阶正则化**的一种极限情形，对应着λ较小时，对两个正值函数的和取最小值

$$\text{使}\quad \chi^2[\mathbf{u}] - \lambda(\mathbf{u}\cdot\mathbf{u})\quad \text{最小} \tag{18.4.11}$$

下面，我们将讨论如何求这样的最小值及一些更一般的情形，而不特别采用SVD。

如果用一个非无穷小的λ值，通过方程(18.4.11)来决定u，将会发生什么情况呢？首先注意，如果$M\gg N$，(未知量数远大于方程数)则u常有足够的自由度使$\chi^2$(方程18.4.9)即使不为0，也会相当不切实际的小。用第15.1节中的话来说，自由度数$\nu=N-M$(当$\nu$很大时约等于$\chi^2$的期望值)，可以减小到零(甚至无意义，超出范围)。但我们知道，对于真实的基础函数，由于没有可调整的参数，自由度数与$\chi^2$的期望值大约应该满足$\nu\approx N$。

增大λ会使解远离$\chi^2$最小的情形，而趋向于使$\mathbf{u}\cdot\mathbf{u}$最小。根据上文对基本点的讨论，我们可以视这种情形为，限制$\chi^2$取某个非零**常值**后使$\mathbf{u}\cdot\mathbf{u}$最小。事实上，一个常见的取法是找到满足$\chi^2=N$的λ值，即通过$\chi^2$的合适取值使问题进一步正则化。求出的$u(x)$称为**零阶正则化反演问题的解**。

$N$值实际上代表所有从均值为$N$，标准差为$(2N)^{1/2}$($\chi^2$的渐近分布)的高斯分布中得到的值。读者可分别试试λ的两个可能值，一个使$\chi^2=N+(2N)^{1/2}$，另一个使$\chi^2=N-(2N)^{1/2}$。

虽然还有更好的方法，但零阶正则化反映了反演问题理论中的经常用到的大部分基本思想。一般有两个正函数，称为$\mathcal{A}$和$\mathcal{B}$。第一个正函数$\mathcal{A}$，是测量模型与数据(例如$\chi^2$)的某种一致性；或者是描述解和基本函数间映射的"鲜明程度"的相关量。当$\mathcal{A}$本身取最小值时，一致性或鲜明性变得很好(通常情形下不可能太好)，但解变得不稳定，剧烈地振荡；或者在另一方面极不现实地反映了由$\mathcal{A}$单独定义一个高退化法的最小值问题。

$\mathcal{B}$正是因此才引入的。它是一个对解的"光滑性"的测量标准；或者有时是一个对应于数据变动使解的稳定性参数化的相关量，或者有时是一个反映解的可能形式的先验判断的量，$\mathcal{B}$被称为**稳定化函数**或**正则化算子**。不论那种情形，$\mathcal{B}$取最小值，均会给出一个或"光滑的"，或"稳定的"，或"可能"的解——与测量数据却毫不相关。

反演理论中唯一的中心思想就是规定

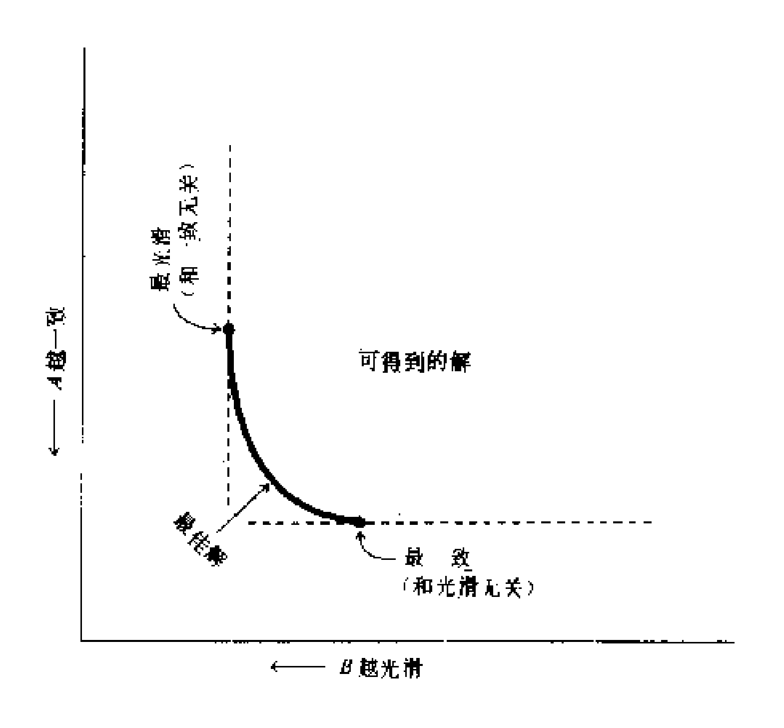
$$\text{使}\quad \mathcal{A}+\lambda\mathcal{B}\quad \text{最小} \tag{18.4.12}$$

其中λ沿所谓交替换位曲线(见图18.4.1)的变化范围是$0<\lambda<\infty$，然后，依据这样或那样的规则停在一个"最佳"λ值上，这可以从相当客观(比如使$\chi^2=N$)变到完全主观。不同的成功方法(我们将讨论其中的几种)的区别表现在：$\mathcal{A}$和$\mathcal{B}$选法的不同；规定式(18.4.12)是满足线性还是非线性方程；λ最终值的多种可能的选择方法以及应用到计算机二维问题(如图像处理)的广泛程度的不同等。

区别还在于它们包含的(或它们的提倡者的)哲学思想的不同。迄今为止，我们一直避免使用"Bayesian"这个词。但要掩盖下述事实却很难，而且我们也不想掩盖，即$\mathcal{B}$与某些**先验**期望，或是先验知识，或是先验解有关；而$\mathcal{A}$则与某些**后验**知识有关，常数λ决定在二者之间一种微妙的协调关系。有些反演方法中"Bayesian"的痕迹较重，但我们认为这完全是历史造成的。一个只注重方程的实际解法而不管它的所包含的哲学思想的人，是很难区分Bayesian方法与其它的经验方法的。

下面三节讨论反演问题的三种不同的方法，它们在不同的领域都有着相当成功的应用。

三种方法都在我们概括出的一般框架内，但每种方法的细节和实施却有很大的区别。




几乎所有的反演问题的方法都涉及两个最优化之间的一个交替换位：数据和解之间的一致性，或者真实解和估算解之间映射的"鲜明程度"（此处用 $\mathscr{A}$ 表示）；与解的光滑性和稳定性（此处用 $\mathscr{B}$ 表示）。图中阴影区表示所有可能的解，那些位于边界上的曲线，是连接 $\mathscr{A}$ 不受限制时的最小值和 $\mathscr{B}$ 不受限制时的最小值的解，它们是"最佳"的解，从某种意义上说，任意其它解被控制在至少有一个解位于曲线上。

图 18.4.1

参考文献和进一步读物：


Craig, I. J. D., and Brow, J. C. 1986, *Inverse Problems in Astronomy*(Bristol, U. K: Adam Hilger).

Tikhonov, A. N., and Goncharsky, A. V. (eds.) 1987, *Ill Posed Problems in the Natural Sciences* (Moscow: MIR).

Parker, R. L. 1977, *Annual Review of Earth and Planetary Science*, vol. 5, pp. 35~64.

Frieden, B. R. 1975, in *Picture Processing and Digital Filtering*, T. S. Huang, ed. (New York: Springer-Verlag).

Tarantola, A. 1987, *Inverse Problem Theory* (Amsterdam: Elsevier).

Baumeister, J. 1987, *Stable Solution of Inverse Problems*(Braunschweig, Germany: Friedr. Vieweg & Sohn) [mathematically oriented].

Titterington, D. M. 1985, *Astronomy and Astrophysics*, vol. 144, pp. 381~387.


## 18.5 线性正则化法

**线性正则化**也称为 Phillips-Twomey 方法[1,2]、**约束线性反演方法**[3]、**正则化方法**[4]或 Tikhonov-Miller **正则化方法**[5]。（也可能还有其它叫法，因为它非常明显地是个好方法。）该方法最简单的形式是零阶正则化（前文方程(18.4.11)）的一个直接推广。和以前一样，函

数 $\mathscr{A}$ 取为 $\chi^2$ 偏差，见方程(18.4.9)，但 $\mathscr{B}$ 由更复杂的从一阶或高阶导数导出的度量光滑性的函数来代替。

例如，假定**先验信息**是：一个可能的 $u(x)$ 和一个常数没有太多的不同，则一个求最小值的较合理的函数应该是

$$\mathscr{B} \propto \int [\hat{u}'(x)]^2 dx \propto \sum_{\mu=1}^{M-1} [\hat{u}_\mu - \hat{u}_{\mu+1}]^2 \tag{18.5.1}$$

因为它非负且仅当 $\hat{u}(x)$ 为常数时才等于零。其中 $\hat{u}_\mu \equiv \hat{u}(x_\mu)$，并且第二个等式(比例式)中假定 $x_\mu$ 是均匀分划的。我们可将 $\mathscr{B}$ 的第二种形式写作

$$\mathscr{B} = |\mathbf{B} \cdot \hat{\mathbf{u}}|^2 = \hat{\mathbf{u}} \cdot (\mathbf{B}^T \cdot \mathbf{B}) \cdot \hat{\mathbf{u}} \equiv \hat{\mathbf{u}} \cdot \mathbf{H} \cdot \hat{\mathbf{u}} \tag{18.5.2}$$

其中 $\hat{\mathbf{u}}$ 是以 $\hat{u}_\mu, \mu=1,\cdots M$ 为分量的向量，$\mathbf{B}$ 是一个 $(M-1)\times M$ 的第一差分矩阵

$$\mathbf{B} = \begin{pmatrix} -1 & 1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & & & & \ddots & & & & \vdots \\ 0 & \cdots & 0 & 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & \cdots & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{pmatrix} \tag{18.5.3}$$

$\mathbf{H}$ 是 $M\times M$ 矩阵

$$\mathbf{H} = \mathbf{B}^T \cdot \mathbf{B} = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & & & & \ddots & & & & \vdots \\ 0 & \cdots & 0 & 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & \cdots & 0 & 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & \cdots & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{pmatrix} \tag{18.5.4}$$

注意 $\mathbf{B}$ 的行数比列小 1。其次，对称的 $\mathbf{H}$ 是退化的；它确实有一个零特征值对应着一个常数函数的值，任意一个常数函数都会使 $\mathscr{B}$ 严格为零。

和第15.4节一样，如果我们记

$$A_{i\mu} \equiv R_{i\mu}/\sigma_i \qquad b_i \equiv c_i/\sigma_i \tag{18.5.5}$$

那么，应用方程(18.4.9)，最小化(18.4.12)变为

$$\text{使} \quad \mathscr{A} + \lambda\mathscr{B} = |\mathbf{A} \cdot \hat{\mathbf{u}} - \mathbf{b}|^2 + \lambda \hat{\mathbf{u}} \cdot \mathbf{H} \cdot \hat{\mathbf{u}} \quad \text{最小} \tag{18.5.6}$$

这可以容易地化简成一个线性正规方程组，仍和第15.4节中一样：解的分量 $\hat{u}_\mu$ 满足 $M$ 个未知量的 $M$ 个方程，

$$\sum_\rho \left[\left(\sum_i A_{i\mu} A_{i\rho}\right) + \lambda H_{\mu\rho}\right] \hat{u}_\rho = \sum_i A_{i\mu} b_i \qquad \mu = 1,2,\ldots,M \tag{18.5.7}$$

或用向量形式为

$$(\mathbf{A}^T \cdot \mathbf{A} + \lambda \mathbf{H}) \cdot \hat{\mathbf{u}} = \mathbf{A}^T \cdot \mathbf{b} \tag{18.5.8}$$

方程(18.5.7)或(18.5.8)可用第二章中介绍标准方法求解，例如 $LU$ 分解法。这时有关正规方程组为病态的常惯注意事项已不再适用，因为 $\lambda$ 项的全部目的是为了消除这相同的病态。但是注意，由于 $\lambda$ 项没有选择一个合适的常数值，所以它本身是病态的。只能希望用户的数据能弥补这一点。

虽然矩阵 $(\mathbf{A}^T\cdot\mathbf{A}+\lambda \mathbf{H})$ 求逆通常并不是求解 $\hat{\mathbf{u}}$ 的最好方法，我们仍将方程(18.5.8)的解先形式上地写成

$$\hat{\mathbf{u}} = \left[\frac{1}{\mathbf{A}^T \cdot \mathbf{A} + \lambda \mathbf{H}} \cdot \mathbf{A}^T \cdot \mathbf{A}\right] \mathbf{A}^{-1} \cdot \mathbf{b} \tag{18.5.9}$$

在其中插进了形式上为 $\mathbf{A}\cdot\mathbf{A}^{-1}$ 的恒等矩阵。因为我们不但将矩阵的逆想象性地写为分母，而且逆矩阵 $\mathbf{A}^{-1}$ 通常并不存在，因此上式只是在形式上成立。但是将方程(18.5.9)与方程(13.6.6)进行最佳或维纳滤波，或者与方程(13.6.6)进行一般的线性预测相比较，都会使我们受到很大的启发。可以看出，$\mathbf{A}^T\cdot\mathbf{A}$ 和 $S^2$ 的作用相同，代表信号功率或自相关性；而 $\lambda\mathbf{H}$ 和 $N^2$ 所起作用相同，代表噪音功率或自相关性。方程(18.5.9)括号中的项有些类似最佳滤波器，其作用是：当 $\mathbf{A}^T\cdot\mathbf{A}$ 足够大时，使病态的逆 $\mathbf{A}^{-1}\cdot\mathbf{b}$ 不修正地通过；而当 $\mathbf{A}^T\cdot\mathbf{A}$ 较小时，则将它抑制掉。

上述 **B** 和 **H** 的选法仅仅是一个明显的导数序列中最简单的情形。如果先验信息是：线性函数，这才是对 u(x)的很好近似，则使下式最小

$$\mathscr{B} \propto \int [\hat{u}''(x)]^2 dx \propto \sum_{\mu=1}^{M-2} [-\hat{u}_\mu + 2\hat{u}_{\mu+1} - \hat{u}_{\mu+2}]^2 \tag{18.5.10}$$

隐含着

$$\mathbf{B} = \begin{pmatrix} -1 & 2 & -1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & & & & \ddots & & & & \vdots \\ 0 & \cdots & 0 & 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & \cdots & 0 & 0 & 0 & 0 & -1 & 2 & -1 \end{pmatrix} \tag{18.5.11}$$

以及

$$\mathbf{H} = \mathbf{B}^T \cdot \mathbf{B} = \begin{pmatrix} 1 & -2 & 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ -2 & 5 & -4 & 1 & 0 & 0 & 0 & \cdots & 0 \\ 1 & -4 & 6 & -4 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & -4 & 6 & -4 & 1 & 0 & \cdots & 0 \\ \vdots & & & & \ddots & & & & \vdots \\ 0 & \cdots & 0 & 1 & -4 & 6 & -4 & 1 & 0 \\ 0 & \cdots & 0 & 0 & 1 & -4 & 6 & -4 & 1 \\ 0 & \cdots & 0 & 0 & 0 & 1 & -4 & 5 & -2 \\ 0 & \cdots & 0 & 0 & 0 & 0 & 1 & -2 & 1 \end{pmatrix} \tag{18.5.12}$$

上式中的 **H** 有两个零特征值，对应于一个线性函数的两个未知确定参数。

如果先验信息是二次函数会更好，则将使下式最小

$$\mathscr{B} \propto \int [\hat{u}'''(x)]^2 dx \propto \sum_{\mu=1}^{M-3} [-\hat{u}_\mu + 3\hat{u}_{\mu+1} - 3\hat{u}_{\mu+2} + \hat{u}_{\mu+3}]^2 \tag{18.5.13}$$

可以得到

$$\mathbf{B} = \begin{pmatrix} -1 & 3 & -3 & 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 3 & -3 & 1 & 0 & 0 & \cdots & 0 \\ \vdots & & & & \ddots & & & & \vdots \\ 0 & \cdots & 0 & 0 & -1 & 3 & -3 & 1 & 0 \\ 0 & \cdots & 0 & 0 & 0 & -1 & 3 & -3 & 1 \end{pmatrix} \tag{18.5.14}$$

及

$$
\mathbf{H} = \begin{pmatrix}
1 & -3 & 3 & -1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
-3 & 10 & -12 & 6 & -1 & 0 & 0 & 0 & 0 & \cdots & 0 \\
3 & -12 & 19 & -15 & 6 & -1 & 0 & 0 & 0 & \cdots & 0 \\
-1 & 6 & -15 & 20 & -15 & 6 & -1 & 0 & 0 & \cdots & 0 \\
0 & -1 & 6 & -15 & 20 & -15 & 6 & -1 & 0 & \cdots & 0 \\
\vdots & & & & & \ddots & & & & & \vdots \\
0 & \cdots & 0 & -1 & 6 & -15 & 20 & -15 & 6 & -1 & 0 \\
0 & \cdots & 0 & 0 & -1 & 6 & -15 & 20 & -15 & 6 & -1 \\
0 & \cdots & 0 & 0 & 0 & -1 & 6 & -15 & 19 & -12 & 3 \\
0 & \cdots & 0 & 0 & 0 & 0 & -1 & 6 & -12 & 10 & -3 \\
0 & \cdots & 0 & 0 & 0 & 0 & 0 & -1 & 3 & -3 & 1
\end{pmatrix} \tag{18.5.15}
$$

(我们将三次方及更高次方的计算留给有兴趣的读者。)

注意，如果读者愿意的话，也可用“向微分方程近似”的方法来正则化，只需令 **B** 为几个适当的有限差分算子(其系数可以依赖于 $x$)的和，并计算 $\mathbf{H}=\mathbf{B}^T\cdot\mathbf{B}$。可以不知道边界条件的值，因为前面已讲过，**B** 的行可以小于列，这样就极有希望用数据来确定它们。当然，如果知道某些边界条件，也可将其加入 **B** 中。

对于上述各式中所有的比例性符号，读者也许不知道应该取 $\lambda$ 为什么实际值首先进行试验。至少一个简单的技巧是可以先试试，令

$$\lambda = \mathrm{Tr}(\mathbf{A}^T\cdot\mathbf{A})/\mathrm{Tr}(\mathbf{H}) \tag{18.5.16}$$

其中 Tr 是矩阵的迹(对角元素之和)。该选法使最小值的两部分有类似的权。因此就可以从这个 $\lambda$ 开始对它们进行调整。

至于 $\lambda$ 的“正确”值是什么，假设已知误差，而且有足够的精度，一个客观的标准便是使 $\chi^2$(即 $|\mathbf{A}\cdot\hat{\mathbf{u}}-\mathbf{b}|^2$)等于测量次数 $N$。上文我们曾提过两个可以接受的孪生选法 $N\pm(2N)^{1/2}$。一个主观的准则是，根据对先验信息的相对依赖程度，选取在 $0<\lambda<\infty$ 范围内的任意喜欢值(人们实际上常常是这样做的，不能指责我们)。

### 18.5.1 二维问题和迭代方法

迄今为止，我们的符号一直是描述一维问题的，例如 $\hat{u}(x)$ 或 $\hat{u}_\mu=\hat{u}(x_\mu)$。不过，所有的讨论都很容易推广到估计一个二维未知量集合 $\hat{u}_{\mu\kappa}$，$\mu=1,\ldots,M$，$\kappa=1,\ldots,K$，的问题，例如，其中 $\hat{u}_{\mu\kappa}$ 对应于一幅测量图像的象素强度。在此情形，我们要解的仍然是方程(18.5.8)。

在图像处理中，一幅“原始”或“受干扰”的测量图像的输入象素数，通常与经过处理后输出的“已知去噪声”的象素数相同，因而矩阵 **R** 和 **A**(方程18.5.5)都是方阵，且均为 $MK\times MK$ 矩阵。**A** 一般很大，不能表示成满阵，但它通常或者是(i)稀疏的，其系数将一个原象素$(i,j)$变模糊为($i\pm$小变化，$j\pm$小变化)，或者是(ii)平移不变性 $A_{(i,j)(\mu,\nu)}=A_{(i-\mu,j-\nu)}$。这两种情形都使问题极容易处理。

在平移不变情形，选用快速傅里叶变换方法(FFT)应该是很明显的。原函数和测量值之间的一般线性关系，现在变成了类似方程(13.1.1)的一个离散卷积。如果用 **k** 表示一个二维波向量，则二维 FFT 将下述变换对之间相互转换：

$$A(i-\mu,j-\nu)\Leftrightarrow\widetilde{\mathbf{A}}(\mathbf{k})\quad b(i,j)\Leftrightarrow\widetilde{b}(\mathbf{k})\quad \hat{u}(i,j)\Leftrightarrow\widetilde{u}(\mathbf{k}) \tag{18.5.17}$$

我们也需要一个正则化或光滑算子 **B** 及由 **B** 导出的 $\mathbf{H}=\mathbf{B}^T\cdot\mathbf{B}$。**B** 的一个流行选法是，拉普拉斯算子的五点有限差分近似，即是每点的值与它笛卡尔坐标四个邻域点的平均值之差。在

傅里叶空间，该选法意味着

$$\widetilde{B}(\mathbf{k}) \propto \sin^2(\pi k_1/M)\sin^2(\pi k_2/K)$$
$$\widetilde{H}(\mathbf{k}) \propto \sin^4(\pi k_1/M)\sin^4(\pi k_2/K) \tag{18.5.18}$$

在傅里叶空间，方程(18.5.7)只是个代数方程，其解为

$$\widetilde{u}(\mathbf{k}) = \frac{\widetilde{A}^*(\mathbf{k})\widetilde{b}(\mathbf{k})}{|\widetilde{A}(\mathbf{k})|^2 + \lambda\widetilde{H}(\mathbf{k})} \tag{18.5.19}$$

其中星号表示复共轭。对于实数据，可利用第12.5节中的FFT程序。

现在，再看A不是平移不变的情形。现在直接解(18.5.8)是不可能的，因为矩阵A太大。我们需要某种迭代方案。

方法之一是，完全用第10.6节中的共轭梯度法来找$\mathscr{A}+\lambda\mathscr{B}$的最小值，见方程(18.5.6)。在第十章的所有方法中，共轭梯度法是唯一最佳的方法，这是因为：(i)它不用存储一个海赛矩阵，该矩阵在此处是求不出来的；(ii)它必须利用梯度信息，而梯度我们很容易求；方程(18.5.6)的梯度是

$$\nabla(\mathscr{A}+\lambda\mathscr{B}) = 2[(\mathbf{A}^T\cdot\mathbf{A}+\lambda\mathbf{H})\cdot\hat{\mathbf{u}} - \mathbf{A}^T\cdot\mathbf{b}] \tag{18.5.20}$$

(参看方程18.5.8)。函数和梯度的估算当然都要利用A为稀疏矩阵的性质，例如通过第2.7节中的**sprsax**和**sprstx**程序。在第18.7节中，我们将要在(非线性)最大熵方法的基础上，进一步讨论共轭梯度法，其中的一些讨论也适用于此。

除共轭梯度法外，可采用不太复杂的快速下降法(见第10.6节)，有时它也能得到很有效的结果，特别是当它与凸集投影联合起来使用时很有效(见下文)。如果$k$次迭代后的解用$\hat{\mathbf{u}}^{(k)}$表示，则经过$k+1$次迭代，我们得到

$$\hat{\mathbf{u}}^{(k+1)} = [\mathbf{1} - \epsilon(\mathbf{A}^T\cdot\mathbf{A}+\lambda\mathbf{H})]\cdot\hat{\mathbf{u}}^{(k)} + \epsilon\mathbf{A}^T\cdot\mathbf{b} \tag{18.5.21}$$

上式中的$\epsilon$是一个反映在下降梯度方向移动大小的参数。当$\epsilon$足够小时上述方法收敛，特别是当$\epsilon$满足

$$0 < \epsilon < \frac{2}{(\mathbf{A}^T\cdot\mathbf{A}+\lambda\mathbf{H})\text{ 的最大特征值}} \tag{18.5.22}$$

寻找$\epsilon$的最佳值或最佳序列有很多复杂的方法，见[7]，或者也可以采用实验方法，对式(18.5.6)进行估算以确保方法的实施是沿下降方向的。

在图像处理问题中，成功与否，在一定程度是主观的(例如"这幅画看起来怎么样?")，因此，迭代式(18.5.21)有时在远未达到收敛前，就会使图像效果有显著的改进。正因为如此，该方法虽然在数学意义上的收敛极其缓慢，但却大量地被应用。实际上，式(18.5.21)可以和$\mathbf{H}=0$一起使用，此时解完全不正则，而且充分收敛时还会产生严重的错误。这种方法称为Van Citter法，可追溯到本世纪三十年代。迭代至1000次的量级并非是不常见的[?]。

### 18.5.2 确定性约束：凸集投影

一些可能的原函数(或图像)集合$\{\hat{\mathbf{u}}\}$，如果该集合中任意两个元素$\hat{\mathbf{u}}_a$和$\hat{\mathbf{u}}_b$的所有线性组合

$$(1-\eta)\hat{\mathbf{u}}_a + \eta\,\hat{\mathbf{u}}_b \qquad 0 < \eta \leq 1 \tag{18.5.23}$$

仍在该集合中，则称该集合为凸的。反演问题中，人们想在解$\hat{\mathbf{u}}$上强制许多确定性的约束，实际上是定义了凸集，例如：

- 正值性
- 紧支柱集(即确定区域外取零值)
- 已知边界(即 $u_L(x) \leqslant \hat{u}(x) \leqslant u_U(x)$,对确定的函数 $u_L$ 和 $u_U$

(在最后一种情形,边界可能和初始估计值及误差规定有关,比如 $\hat{u}_0(x) \pm \gamma\sigma(x)$,其中 $\gamma$ 是 1 或 2。)注意,这些以及其它类似的限制可加在图像空间或傅里叶变换空间,也可(实际上)加在 $\hat{\mathbf{u}}$ 的任意线性变换的空间。

假定 $C_i$ 是一个凸集,那 $\mathscr{P}_i$ 称为到该集的**非扩张投影算子**,如果(i)$\mathscr{P}_i$ 将任意已在 $C_i$ 中的 $\hat{\mathbf{u}}$ 保留不变,(ii)$\mathscr{P}_i$ 将 $C_i$ 中的任意点 $\hat{\mathbf{u}}$ 在下述意义下映射到 $C_i$ 中和 $\hat{\mathbf{u}}$ 距离最近的一个元素,即

$$|\mathscr{P}_i\hat{\mathbf{u}} - \hat{\mathbf{u}}| \leq |\hat{\mathbf{u}}_a - \hat{\mathbf{u}}| \qquad \text{对所有 } C_i \text{ 中的 } \hat{\mathbf{u}}_a \tag{18.5.24}$$

该定义虽然看来复杂,实际例子却很简单。正 $\hat{\mathbf{u}}$ 集的一个非扩张投影算子,"将 $\hat{\mathbf{u}}$ 的所有负分量映为零"。以 $u_L(x) \leq \hat{u}(x) \leq u_U(x)$ 为边界的 $\hat{u}(x)$ 集的一个非扩张投影算子,"将所有小于下界的值映为下界,而将所有大于上界的值映为上界"。具有紧支柱集的函数集的一个非扩张投影算子,"将所有支柱集区域外的值映为零"。

这些定义对下述著名定理十分有用:令 $C$ 为 $m$ 个凸集 $C_1, C_2, \cdots, C_m$ 的交集,则迭代

$$\hat{\mathbf{u}}^{(k+1)} = (\mathscr{P}_1\mathscr{P}_2\cdots\mathscr{P}_m)\hat{\mathbf{u}}^{(k)} \tag{18.5.25}$$

当 $k \to \infty$ 时,将从所有起始点收敛到 $C$ 中。而且,如果 $C$ 为空集(无交集),那么这个迭代没有极限点。该定理的实际应用被称为**到凸集的投影方法**,或有时简写为 POCS[7]。

POCS 定理的一般形式是用一组 $\mathscr{T}_i$ 来代替 $\mathscr{P}_i$,

$$\mathscr{T}_i = 1 + \beta_i(\mathscr{P}_i - 1) \qquad 0 < \beta_i < 2 \tag{18.5.26}$$

适当选取 $\beta_i$ 集可以加快到交集 $C$ 的收敛速度。

有些反演问题,可以只通过迭代式(18.5.25)就可完全解决。例如,在天文成像及 $X$ 射线衍射工作中,经常遇到的问题是要求恢复图象,但只给出图象傅叶变换的**模**(相当于光谱强度或自相关值)而不给出**相**。这时,就可以利用下述三个凸集:其傅里叶变换在确定的误差范围内有确定的模的所有图像的集合;所有正图像的集合;以及某特定区域外强度为零的所有图象的集合。在这种情形下,POCS 迭代(18.5.25)在这三个集合之间循环,轮流施加每种约束;每当施加傅里叶约束时,就用 FFT 将其映入映出傅里叶空间。

POCS 在空间域和傅里叶域交替施加约束的特定应用被称为 Gerchberg-Saxton 算法[8]。该算法虽然是非扩张的,而且实际上也常常是收敛的,但是还没有证明在所有情形下都收敛[9]。在上面提到的相恢复问题中,该算法在取得突然的、明显的改进之前,常常有许多次迭代都"粘滞"在一个平稳阶段。有时,需要进行多达 $10^4$ 到 $10^5$ 次迭代。(对于"非粘滞"过程,见[10])。虽然我们相信有一定复杂度的二维图像的解是唯一的,但有关解的唯一性问题研究得还不十分清楚。

通过投影算子,确定性约束可以合并到线性正则化迭代方法中。特别地,改变各项的组合,我们可将迭代(18.5.21)写为

$$\hat{\mathbf{u}}^{(k+1)} = [\mathbf{1} - \epsilon\lambda\,\mathbf{H}] \cdot \hat{\mathbf{u}}^{(k)} + \epsilon\mathbf{A}^T \cdot (\mathbf{b} - \mathbf{A} \cdot \hat{\mathbf{u}}^{(k)}) \tag{18.5.27}$$

如果迭代在每步都加上投影算子

$$\hat{\mathbf{u}}^{(k+1)} = (\mathscr{P}_1\mathscr{P}_2\cdots\mathscr{P}_m)[\mathbf{1} - \epsilon\lambda\,\mathbf{H}] \cdot \hat{\mathbf{u}}^{(k)} + \epsilon\mathbf{A}^T \cdot (\mathbf{b} - \mathbf{A}\hat{\mathbf{u}}^{(k)}) \tag{18.5.28}$$

(或用方程18.5.26的算子 $\mathcal{D}_i$ 代替 $\mathcal{B}_i$)，则我们可以看到收敛条件(18.5.22)并未改变，而且在施加期望的非线性约束后，迭代将收敛到使二次函数(18.5.6)最小。对于更复杂的情形以及和迭代有关的更快的收敛，参考[7]。


参考文献和进一步读物：

Phillips, D. L. 1962, *Journal of the Association for Computing Machinery*, vol. 9, pp. 84～97. [1]

Twomey, S. 1963, *Journal of the Association for Computing Machinery*, vol. 10, pp. 97～101. [2]

Twomey, S. 1977, *Introduction to the Mathematics of inversion in Remote Sensing and Indirect Measurements* (Amsterdam: Elsevier). [3]

Craig, I. J. D., and Brown, J. C. 1986, *Inverse Problems in Astronomy* (Bristol, U. K.: Adam Hilger). [4]

Tikhonov, A. N., and Arsenin, V. Y. 1977, *Solutions of Ill-Posed Problems* (New York: Wiley). [5]

Tikhonov, A. N., and Goncharsky, A. V. (eds). 1987, *Ill-Posed Problems in the Natural Sciences* (Moscow: MIR).

Miller, K. 1970, *SIAM Journal on Mathematical Analysis*, vol. 1, pp. 52～74. [6]

Schafer, R. W., Mersereau, R. M., and Richards, M. A. 1981, *Proceedings of the IEEE*, vol. 69, pp. 432～450.

Biemond, J., Lagendijk, R. L., and Mersereau, R. M. 1990, *Proceedings of the IEEE*, vol. 78, pp. 856～883. [7]

Gerchberg, R. W., and Saxton, W. O. 1972, *Optik*, vol. 35, pp. 237～246. [8]

Fienup, J. R. 1982, *Applied Optics*, vol. 15, pp. 2758～2769. [9]

Fienup, J. R., and Wackerman, C. C. 1986, *Journal of the Optical Society of America A*, vol. 3, pp. 1897～1907. [10]


## 18.6 柏克斯-吉尔伯特法

柏克斯-吉尔伯特(Backus-Gilbert)方法[1,2](摘要见[3]或[4])与其它正则化方法的区别在于，它的函数 $\mathcal{A}$ 和 $\mathcal{B}$ 具有不同的性质。对于 $\mathcal{B}$，该方法力图使解 $\hat{u}(x)$ 的**稳定性**最好，而不是象第一种情形那样注重光滑性。即

$$\mathcal{B} \equiv \mathrm{Var}[\hat{u}(x)] \tag{18.6.1}$$

是一个用来度量数据在测量误差范围内变化时解的变化程度。注意，该方差不是 $\hat{u}(x)$ 与真实 $u(x)$ 之间预期的偏差——这个偏差由 $\mathcal{A}$ 来约束——而是在进行多次重复实验时，测量 $u(x)$ 的估计值在预期的实验和实验之间的分散程度。

对于 $\mathcal{A}$，柏克斯-吉尔伯特方法着眼于，解 $\hat{u}(x)$ 与真实函数 $u(x)$ 之间的关系，力图使二者之间的映射，在无差错的数据范围内，尽可能地接近于恒等映射。该方法是线性的，因而 $\hat{u}(x)$ 和 $u(x)$ 之间的关系可以写为

$$\hat{u}(x) = \int \hat{\delta}(x,x')\, u(x')dx' \tag{18.6.2}$$

其中 $\hat{\delta}(x,x')$ 是所谓的**分辨率函数**或**平均核**。柏克斯-吉尔伯特方法力图使 $\hat{\delta}$ 的宽度或散布度最小(即使分辨率最大)。而 $\mathcal{A}$ 选作衡量散布度的某个正数。

虽然柏克斯-吉尔伯特的哲学思想与菲力普斯-吐梅(*Phillips-Twomey*)的哲学思想及

其相关联的方法有很大的差别，但在实际运用中这一差别却比我们想象的要小得多。一个稳定解几乎毫无例外的是光滑的：一个非正则化解产生的剧烈的，不稳定的振动对数据的小变化常常十分敏感。同样，使 $\hat{u}(x)$ 接近于 $u(x)$ 将肯定使无差错的数据与模型一致。因而 $\mathscr{A}$ 和 $\mathscr{B}$ 所起的作用与前两节相应的起的作用极其相似。

柏克斯-吉尔伯特公式的主要优点是，它能很好地控制它所注重测量的那些项的性质，例如稳定性和分辨率。而且在处理任何实际数据之前，λ(与它通常的作用一样，用来协调 $\mathscr{A}$ 与 $\mathscr{B}$)都有常规的取法，或至少很容易选取。因而避免了选取一个 λ 所产生的困难及其潜在的主观偏见。在需要进行数据转换的实验的设计与预测之类问题中，常可考虑选用柏克斯-吉尔伯特方法。

让我们来看看这一切是怎样做的。从方程(18.4.5)开始，

$$c_i \equiv s_i + n_i = \int r_i(x)u(x)dx + n_i \tag{18.6.3}$$

并且从一开始就建立线性关系，我们找一组**反演响应核** $q_i(x)$ 满足

$$\hat{u}(x) = \sum_i q_i(x)c_i \tag{18.6.4}$$

并期望能用它来估计 $u(x)$。定义每个数据点的响应核的积分也很有用，

$$R_i \equiv \int r_i(x)dx \tag{18.6.5}$$

将方程(18.6.4)代入方程(18.6.3)，并与方程(18.6.2)作比较，我们发现

$$\hat{\delta}(x,x') = \sum_i q_i(x)r_i(x') \tag{18.6.6}$$

我们能够要求对每个 $x$ 使平均核具有单位面积，从而给出

$$1 = \int\hat{\delta}(x,x')dx' = \sum_i q_i(x)\int r_i(x')dx' = \sum_i q_i(x)R_i \equiv \mathbf{q}(x)\cdot\mathbf{R} \tag{18.6.7}$$

其中 $\mathbf{q}(x)$ 和 $\mathbf{R}$ 都是具有 $N$ 个分量的向量，$N$ 为测量数。

误差的标准传播及方程(18.6.1)给出

$$\mathscr{B} = \text{Var}[\hat{u}(x)] = \sum_i\sum_j q_i(x)S_{ij}q_j(x) = \mathbf{q}(\mathbf{x})\cdot\mathbf{S}\cdot\mathbf{q}(x) \tag{18.6.8}$$

其中 $S_{ij}$ 是协方差矩阵(方程(18.4.6))。如果非对角线的协方差可以忽略(相当于 $c_i$ 的误差是独立的)，则 $S_{ij}=\delta_{ij}\sigma_i^2$ 是对角矩阵。

现在，我们需要对每个 $x$ 的取值，定义一个 $\delta(\mathbf{x},\mathbf{x}')$ 的宽度或散布度的测量标准。虽然有多种可能的选取，但柏克斯和吉尔伯特选择了平方的二阶矩。该测量标准就是函数 $\mathscr{A}$

$$\begin{aligned}\mathscr{A} = w(x) &= \int(x'-x)^2[\hat{\delta}(x,x')]^2dx' \\ &= \sum_i\sum_j q_i(x)W_{ij}(x)q_j(x) \equiv \mathbf{q}(x)\cdot\mathbf{W}(x)\cdot\mathbf{q}(x)\end{aligned} \tag{18.6.9}$$

这里我们使用了方程(18.6.6)并将散布矩阵 $\mathbf{W}(x)$ 定义为

$$W_{ij}(x) \equiv \int(x'-x)^2r_i(x')r_j(x')dx' \tag{18.6.10}$$

现在函数 $q_i(x)$ 通过最小化原则来确定：

$$\text{使}\quad \mathscr{A}+\lambda\mathscr{B} = \mathbf{q}(x)\cdot[\mathbf{W}(x)+\lambda\,\mathbf{S}]\cdot\mathbf{q}(x)\quad\text{最小} \tag{18.6.11}$$

并服从约束条件(18.6.7)式，即 $\mathbf{q}(x)\cdot\mathbf{R}=1$。

方程(18.6.11)的解是

$$\mathbf{q}(x) = \frac{[\mathbf{W}(x) + \lambda \mathbf{S}]^{-1} \cdot \mathbf{R}}{\mathbf{R} \cdot [\mathbf{W}(x) + \lambda \mathbf{S}]^{-1} \cdot \mathbf{R}} \tag{18.6.12}$$

(参考文献[4]给了一个很容易接受的证明。)对于任意特定的数据集C(测量值$c_i$的集合),解$\hat{u}(x)$为

$$\hat{u}(x) = \frac{\mathbf{C} \cdot [\mathbf{W}(x) + \lambda \mathbf{S}]^{-1} \cdot \mathbf{R}}{\mathbf{R} \cdot [\mathbf{W}(x) + \lambda \mathbf{S}]^{-1} \cdot \mathbf{R}} \tag{18.6.13}$$

(不要被符号所迷惑,误以为是对整个矩阵$\mathbf{W}(x)+\lambda\mathbf{S}$求逆,只要解线性系统$(\mathbf{W}(x)+\lambda\mathbf{S})\cdot\mathbf{y}=\mathbf{R}$求出$\mathbf{y}$,然后将$\mathbf{y}$代入式(18.6.12)或(18.6.13)的分子与分母即可)。

方程(18.6.12)、(18.6.13)与(18.5.7)、(18.5.8)的线性正则化解有着完全不同的特点。方程(18.6.12)中的向量和矩阵的大小都为$N$,$N$为测量数;基础变量$x$没有离散化,因而$M$不会出现;又对每预期的$x$值,都必须解不同的$N\times N$线性方程组集。而在式(18.5.8)中,需解$M\times M$线性方程组,但只需解一次。一般说来,反复解线性方程组这个计算上的负担使柏克斯-吉尔伯特方法对一维以外的其他问题均不适用。

柏克斯-吉尔斯特方法的$\lambda$如何选取呢?前面曾提过,在掌握任何实际数据前就能(有时是应该)对$\lambda$作出选择。对给定的一个$\lambda$试验值及$x$的一个序列,先利用方程(18.6.12)计算$\mathbf{q}(x)$;然后再运用方程(18.6.6)绘制出分辨率函数$\hat{\delta}(x,x')$为$x'$的函数,这个绘图中对应$x'$的幅度,将反映不同的$x'$值对那点$\hat{u}(x)$估计值所起的作用的大小。对同一$\lambda$值,利用方程(18.6.8)再绘制出函数$\sqrt{\mathrm{Var}[\hat{u}(x)]}$。(这需要对协方差矩阵的测量值先做出估计)。

当$\lambda$变动时,就会清楚地看到分辨率和稳定性之间的交替换位。这时挑选所需要的值。假如读者愿意的话,方程(18.6.12)和(18.6.13)中的$\lambda$甚至可以取选为$x$的函数,即$\lambda=\lambda(x)$。(对每个$x$值都必须求解独立的一组方程这是一个优点)通过对$\lambda$值的选取,现在读者已对反演求解过程有了定量的认识。一旦进行真实数据的处理,需要判断某个特定的性质,例如一个尖峰或突变是否真实,和(或)实际上是否可解,这时上述做法就极其有用。柏克斯-吉尔伯特法在地球物理学中已得到了极其成功的应用,用该方法可从地震数据中求取了地球结构的信息(比如,密度随深度的变化)。

参考文献和进一步读物:


Backus, G. E., and Gilbert, F. 1968, *Geophysical Journal of the Royal Astronomical Society*, vol. 16, pp. 169~205. [1]

Backus, G. E., and Gilbert, F. 1970, *Philosophical Transactions of the Royal Society of London A* vol. 266, pp. 123~192. [2]

Parker, R. L. 1977, *Annual Review of Earth and Planetary Science*, vol. 5, pp. 35~64. [3]

Loredo, T. J., and Epstein, R. I. 1989, *Astrophysical Journal*, vol. 336, pp. 896~919. [4]


## 18.7 最大熵图像恢复

前面我们曾说过,某些反演方法与贝叶斯观点的结合在很大程度上是历史的偶然,而并非是理智的必然。所谓的**最大熵方法**就是该结合的著名产物;对这些反演方法的概括总结如果不引用一些(至少是引论性的)贝叶斯观点,将会变得黯然失色。我们同时也应该指出,这

里讲的最大熵反演方法与第13.7节中讨论过的最大熵谱估计之间的联系是非常没有实际意义的。虽然二者都称为**最大熵方法**或 **MEM**，但实际应用中，两种方法并无联系。

从概率论基本公理推出的贝叶斯定理描述了两个事件，比如 A 和 B 之间的条件概率：

$$\mathrm{Prob}(A|B)=\mathrm{Prob}(A)\frac{\mathrm{Prob}(B|A)}{\mathrm{Prob}(B)} \tag{18.7.1}$$

其中 $\mathrm{Prob}(A|B)$ 是度量了事件 $B$ 发生后 $A$ 发生的概率，$\mathrm{Prob}(B|A)$ 类似，而 $\mathrm{Prob}(A)$ 和 $\mathrm{Prob}(B)$ 为绝对概率。

"贝叶斯论者"(不妨这样称呼)比所谓的"频率论者"对概率采纳了一种更广泛的解释。贝叶斯论者认为 $\mathrm{Prob}(A|B)$ 是度量了事件 $B$ 发生后，$A$ 事件发生的可能性的大小，其值变化范围从 0 到 1。在这个较广泛的观点下，$A$ 和 $B$ 不一定为可重复事件，它们可以是命题或假设。于是概率论的公式变成了一组处理推理的相容规则[1,2]。由于可能性本身常常是以一些也许不能明确表达的假设为条件的，因而所有贝叶斯概率都被看作是，以一些整体的背景信息 I 为条件的。

假定 $H$ 为某假设。即使在取得在任何直接的数据之前，贝叶斯论者也能赋于 $H$ 一定程度的可能性 $\mathrm{Prob}(H|I)$，称为"贝叶斯先验"。当数据 $D_1$ 出现后，贝叶斯定理告诉我们如何重新估计 $H$ 的可能性：

$$\mathrm{Prob}(H|D_1I)=\mathrm{Prob}(H|I)\frac{\mathrm{Prob}(D_1|HI)}{\mathrm{Prob}(D_1|I)} \tag{18.7.2}$$

方程(18.7.2)右端分子中的因子，是给定假设后数据集的概率，它是可计算的(与第15.1节"可能性"比较)。分母被称为数据的"先验预测概率"，在这情形下它不过是一个规一化常数，它可以根据所有假设的概率之和应该为1这个性质来计算出。(在其它一些贝叶斯问题中，可用的两个完全不同模型的先验预测概率来估计它们的相对可能性。)

如果第二天又获得数据 $D_2$，我们还可将 $H$ 的概率估计值进一步修正为

$$\mathrm{Prob}(H|D_2D_1I)=\mathrm{Prob}(H|D_1I)\frac{\mathrm{Prob}(D_2|HD_1I)}{\mathrm{Prob}(D_2|D_1I)} \tag{18.7.3}$$

利用概率的乘积法则，$\mathrm{Prob}(AB|C)=\mathrm{Prob}(A|C)\mathrm{Prob}(B|AC)$，我们发现方程(18.7.2)和(18.7.3)意味着

$$\mathrm{Prob}(H|D_2D_1I)=\mathrm{Prob}(H|I)\frac{\mathrm{Prob}(D_2D_1|HI)}{\mathrm{Prob}(D_2D_1|I)} \tag{18.7.4}$$

它表明如果所有的数据同时取得时，我们将得到相同的结果。

从贝叶斯论者的观点来看，反演问题就是推理问题[3,4]。基本参数集 $\mathbf{u}$ 是一个假设，在给定测量数据值 $\mathbf{c}$ 后，$\mathbf{u}$ 的概率及贝叶斯先验 $\mathrm{Prob}(\mathbf{u}|I)$ 都可以计算出。也许我们想找一个单一的"最佳"的反演 $\mathbf{u}$，使下式

$$\mathrm{Prob}(\mathbf{u}|\mathbf{c}I)=\mathrm{Prob}(\mathbf{c}|\mathbf{u}I)\frac{\mathrm{Prob}(\mathbf{u}|I)}{\mathrm{Prob}(\mathbf{c}|I)} \tag{18.7.5}$$

在 $\mathbf{u}$ 的所有可能选择下最大。贝叶斯分析也允许记录额外信息可能性的存在，该额外信息刻划了 $\mathbf{u}$ 的具有较高的相对概率的可能区域，即所谓 $\mathbf{u}$ 的"后验泡"。

给定假设 $\mathbf{u}$，数据 $\mathbf{c}$ 的概率的计算与最大似然法中的步骤完全相同。例如，对于高斯误差，它由下式给出

$$\mathrm{Prob}(\mathbf{c}|\mathbf{u}I)=\exp\left(-\frac{1}{2}\chi^2\right)\Delta u_1\Delta u_2\ldots\Delta u_M \tag{18.7.6}$$

其中，$\chi^2$可以采用方程(18.4.9)从 u 和 c 计算得到，并且所有 $\Delta u_\mu$ 均为 $u$ 分量的小的常数区域，但这些分量的实际大小无关紧要，因为它们并不依赖于 u(比较方程(15.1.3)和(15.1.4))。

在最大似然估算中，我们实际上选取先验 Prob(u|I)为常数。当从大批数据中估计少量的几个参数时，该方法对我们来说很不经济。这里，"参数"(u 的分量)的个数与测量(c 的分量)的个数可相比拟或较大些。并且我们还需要找一个非平凡的先验 Prob(u|I)来解决解的退化性问题。

在最大熵图像恢复中，熵这个概念出现了。一个物理系统在宏观状态下的熵，常用 $S$ 表示，是在微观状态下不同结构数的对数，这些微观结构都具有同样的宏观可观测值(即与观察到的宏观状态一致)。实际上，我们会发现熵的负数也很有用，或称为**负熵**，表示为 $H=-S$(该表示法可以追溯到波尔兹曼。在有充分的理由确信微观结构的先验概率都相同的情况下(这种情形称为各态历经的)，一个熵为 $S$ 的宏观状态的贝叶斯先验 Prob(u|I)与 exp($S$)或 exp($-H$)成比例。

MEM 利用这个原理，对任意给定的基本函数 u 赋于一个先验概率。例如[7]，假定每个象素亮度的测量值可量化(以某种单位)成一个整数值。令

$$U=\sum_{\mu=1}^{M}u_\mu \tag{18.7.7}$$

为整个图像中亮度的量化值总数，那么，我们可根据下述思想来确定我们的"先验"，即每个亮度量级在任意象素上，都有相等的先验可能性。(对于该思想更抽象的一些应用，见[8])。因此，获得一个特定结构 u 的方法的个数为

$$\frac{U!}{u_1!u_2!\ldots u_M!}\propto\exp\left[-\sum_\mu u_\mu\ln(u_\mu/U+\frac{1}{2}(\ln U-\sum_\mu \ln u_\mu)\right] \tag{18.7.8}$$

上式左端可理解为，亮度量化值总数的不同顺序排列的数目，除以每个象素内等价的重新排序的数目；而上式右端是通过阶乘函数的斯特林(Stirling)近似得到的。取负对数，并且在阶为 $U$ 项出现处，忽略阶为 $\log U$ 的项，我们得到负熵为

$$H(\mathbf{u})=\sum_{\mu=1}^{M}u_\mu\ln(u_\mu/U) \tag{18.7.9}$$

现在，我们根据方程(18.7.5)、(18.7.6)和(18.7.9)使下式最大

$$\text{Prob}(\mathbf{u}|\mathbf{c})\propto\exp\left[-\frac{1}{2}\chi^2\right]\exp[-H(\mathbf{u})] \tag{18.7.1}$$

或等价地

$$\text{使}\quad \ln[\text{Prob}(\mathbf{u}|\mathbf{c})]=\frac{1}{2}\chi^2[\mathbf{u}]+H(\mathbf{u})=\frac{1}{2}\chi^2[\mathbf{u}]+\sum_{\mu=1}^{M}u_\mu\ln(u_\mu/U) \tag{18.7.11}$$

这应当使读者想起方程(18.4.11)或方程(18.5.6)，或我们实际上讨论过的沿曲线 $\mathscr{A}+\lambda\mathscr{B}$ 的任意一个最小化原则，其中 $\lambda\mathscr{B}=H(\mathbf{u})$是一个正则化算子。$\lambda$ 怎么取？根据在方程(18.4.11)后讨论过的理由：退化反演可能会得到一个 $\chi^2$的不切实际的小值。我们需要一个可调整的参数，使 $\chi^2$落在预期的统计意义上的狭小区域 $N\pm(2N)^{1/2}$中。第18.4节中开始的讨论表明：我们将 $\lambda$ 归入哪一项并没有区别。为了保持符号上的一致性，我们在 $\lambda$ 中吸收一个因子 2，并将其归入熵项。(一个不确定的 $\lambda$ 因子的必要性还可从以下得知：如果在改变 u

的量化单位时,例如一个 8 位模数转换器用一个 12 位模数转换器来替换时,我们的最小化原则应保持不变,则可看出 $\lambda$ 因子是必需的)。现在,我们也可以用“前面的说法”来表明,这个过程正是获得所选择的统计估算器的过程,它

$$使\quad \mathcal{A}+\lambda\mathcal{B}=\chi^2[\hat{\mathbf{u}}]+\lambda H[\hat{\mathbf{u}}]=\chi^2[\hat{\mathbf{u}}]+\lambda\sum_{\mu}^{M}\hat{u}_\mu\ln(\hat{u}_\mu) \tag{18.7.12}$$

(我们也可以添加一项拉格朗日乘子乘 $\lambda' U$,以限制总强度 $U$ 为常数)。

不难看出,负熵 $H(\hat{\mathbf{u}})$ 实际上是一个正则化算子,与 $\hat{\mathbf{u}}\cdot\hat{\mathbf{u}}$(方程(18.4.11))及 $\hat{\mathbf{u}}\cdot\mathbf{H}\cdot\hat{\mathbf{u}}$(方程(18.5.6))类似。它的下述性质很值得注意:

1. 当 $U$ 保持为常数时,对于 $\hat{u}_\mu=U/M=$ 常数,使 $H(\hat{\mathbf{u}})$ 最小。因而在希望得到一个常数解的意义下,$H(\hat{\mathbf{u}})$ 是光滑的,这与方程(18.5.4)类似。常数解是一个最小值,这个结论是根据 $u\ln u$ 的二阶导数为正值而得到的。
2. 但是与方程(18.5.4)不同,$H(\hat{\mathbf{u}})$ **是局部的**,即对邻近的象素没有区分。它只是简单地将某函数 $f$ 累加起来,此处在所有象素上,函数为

$$f(u)=u\ln u \tag{18.7.13}$$

   事实上,当一幅图像中的象素被完全量化时,$H(\hat{\mathbf{u}})$ 是个不变量。这种形式表明,在低强度光滑背景上,嵌入少量极亮的象素(点源时),$H(\hat{\mathbf{u}})$ 并不显著增加。
3. 当任意一个象素为零时,$H(\hat{\mathbf{u}})$ 将无限变陡。这一点就强制地使图像保持正值性,这样就不再需要额外的确定性约束。
4. $H(\hat{\mathbf{u}})$ 与我们见过的其它正则化算子之间的最大区别是,$H(\hat{\mathbf{u}})$ 不是 $\hat{\mathbf{u}}$ 的二次函数。因而在变化方程(18.7.12)时,所得到的方程仍是非线性的。这个事实本身就值得进行进一步讨论。

非线性方程比线性方程难解。但是对于图像处理问题,通常有大量的方程需要用迭代过程来求解,既使这些方程是线性的。因而,非线性因素的实际效果得到了缓解。下面,我们将概括地总结一些 MEM 反演问题中成功应用的方法。

对某些问题,其中一个著名的例子是射电天文学中,根据一组完整的傅里叶系数来进行图像恢复的问题,MEM 反演的高级实施甚至还可以部分地探测出 $H(\hat{\mathbf{u}})$ 的非线性特征。方法之一[6]是,考虑正确测量值的极限 $\sigma_i\to 0$。这时,最小化则原则(18.7.12)中的 $\chi^2$ 项被一组约束所替代,其中每个约束都有它自己的拉格朗日乘子,这都为了使模型和数据之间保持一致性,即

$$使\quad \sum_j\lambda_j\Big[c_j-\sum_\mu R_{j\mu}\hat{u}_\mu\Big]+H(\hat{\mathbf{u}})\quad 最小 \tag{18.7.14}$$

(参看方程18.4.7。)设对 $\hat{u}_\mu$ 的法向导数为零,可得

$$\frac{\partial H}{\partial \hat{u}_\mu}=f'(\hat{u}_\mu)=\sum_j\lambda_j R_{j\mu} \tag{18.7.15}$$

或定义一个函数 $G$ 为 $f'$ 的逆函数,

$$\hat{u}_\mu=G\Big(\sum_j\lambda_j R_{j\mu}\Big) \tag{18.7.16}$$

上式中的解只是形式上的,因为必须先求出 $\lambda_j$,而它根据使方程(18.7.16)满足所有添加到式(18.7.14)中的约束条件找出来的。但是,方程(18.7.16)也的确表明了下述重要事实,即如果 $G$ **是线性的**,那么解 $\hat{\mathbf{u}}$ 只包含基函数 $R_{j\mu}$ 的一个线性组合,其中 $R_{j\mu}$ 对应着实际测量值

$j$。这相当于设置未测量的 $c_j$ 为零。注意，从方程(18.4.11)得到的基本解实际上就有一个线性的 $G$。

在不完整的傅里叶图像重建问题中，典型的 $R_{j\mu}$ 有 $\exp(-2\pi i\mathbf{k}_j\cdot\mathbf{x}_\mu)$ 这种形式，其中 $\mathbf{x}_\mu$ 是图像空间的一个二维向量，$\mathbf{k}_\mu$ 是一个二维波向量。如果图像包含强点源，那么设置未测量的 $c_j$ 为零的效果，就等于在整个图像平面产生一些单侧波纹。这些波纹会掩盖了位于强点源之间原有的低强度图像的特征。但是，如果 $G$ 的斜率在自变量数值小时会更小，在自变量数值大时更大，那么，图像低强度部分的波纹相对地受到抑制，而强点源相对地将变尖锐("超解")。$G$ 的斜率的这种性质等价于要求 $f'''(u)<0$。对于函数 $f(u)=u\ln u$，我们确实有 $f'''=-1/u^2<0$。

用更形象化的语言来说，非线性用来"产生"未测量的 $c_j$ 的非零值，从而使低强度的波纹受到抑制，并使点源变得尖锐。

### 18.7.1 MEM 真的不可思议吗?

负熵函数(18.7.9)的唯一性怎么样?注意，那个方程的前提是假设亮度元素**先验**地均匀分布到所有象素上。如果我们想得到另外一种**先验**图像，比如象素强度为 $m_\mu$，则容易证明负熵函数变成了

$$H(\mathbf{u}) = \sum_{\mu=1}^{M} u_\mu \ln(u_\mu/m_\mu) - \text{常数} \tag{18.7.17}$$

(常数可以忽略。)然后就可以进行其余的讨论。

更重要的是，尽管有许多人极力提倡[7]，但函数形式 $f(u)=u\ln u$ 实际上并不普遍成立。在其它一些物理现象中(例如，在射电天文中，每种模式有许多光量子的限制情况下，一个电磁场的熵)，负熵函数实际上是 $f(u)=-\ln u$(其它例子见[8])，一般而言，"熵是什么"这个问题，在任意特定的情形下并没有唯一的答案(对于在适当条件下，寻找一种更普遍的规则来使熵函数具有某种形式，人们做了很多努力，请参阅文献[9])。

上面总结的四条性质，加上非线性的所需符号，$f'''(u)<0$，则对函数 $f(u)=-\ln u$ 及 $f(u)=\ln u$ 都是适用的。事实上，一个很简单的非线性函数 $f(u)=-\sqrt{u}$ 也具有上述性质，但这个函数并不能在理论上确定出任何信息(没有算法!)。利用这些熵函数形式，对试验图像进行 MEM 重建，实际上没有任何差别。

所有已知的证据表明，MEM 只不过是迄今为止我们所讨论过的众多的一般正则化方法 $\mathscr{A}+\lambda\mathscr{B}$ 中的一种。当将 MEM 应用到根据不完整傅里叶数据进行的图像重建，并且已估计到这些图像除有许多极亮点源外，还包括一些有趣的低强度分布源时，它的独特性使得它对该问题的处理变得强劲有力。但对于具其它性质的图像，没有理由说 MEM 方法通常比其它正则化方法更好，不管这些正则化方法是我们已知的还是未知的。

### 18.7.1 *MEM* 的算法

目的是寻找向量 $\hat{\mathbf{u}}$，使 $\mathscr{A}+\lambda\mathscr{B}$ 最小，其中各表达式见方程(18.5.5)、(18.5.6)及(18.7.13)，

$$\mathscr{A} = |\mathbf{b}-\mathbf{A}\cdot\hat{\mathbf{u}}|^2 \quad \mathscr{B} = \sum_\mu f(\hat{u}_\mu) \tag{18.7.18}$$

与“一般”的最小化问题相比，我们具有可以直接计算梯度及二阶偏导数矩阵（海赛矩阵）这个有利条件，

$$\nabla\mathscr{A} = 2(\mathbf{A}^T\cdot\mathbf{A}\cdot\hat{\mathbf{u}} - \mathbf{A}^T\cdot\mathbf{b}) \qquad \frac{\partial^2\mathscr{A}}{\partial\hat{u}_\mu\,\partial\hat{u}_\rho} = [2\mathbf{A}^T\cdot\mathbf{A}]_{\mu\rho}$$

$$[\nabla\mathscr{B}]_\mu = f'(\hat{u}_\mu) \qquad \frac{\partial^2\mathscr{B}}{\partial\hat{u}_\mu\,\partial\hat{u}_\rho} = \delta_{\mu\rho}f''(\hat{u}_\mu) \tag{18.7.19}$$

重要的是要注意，虽然 $\mathscr{A}$ 的二阶偏导数矩阵不能存储（这矩阵的大小是象素数的平方），但它可以先通过 $\mathbf{A}$，再通过 $\mathbf{A}^T$ 作用到任意向量上。在根据不完整的傅里叶数据进行图像重建的情况中，或在与一个平移不变点分布函数作卷积的情况中，这些应用将包括几个 FFT。同样，在采用 $\mathbf{A}$ 和 $\mathbf{A}^T$ 的作用方法下，梯度 $\nabla\mathscr{A}$ 的计算也将包括几个 FFT。

虽然经典的共轭梯度法（第10.6节）取得了一些成功的应用，但是 $f(u)=u\ln u$ 中的非线性常常会使该方法受到很大的影响。如果试验步长使得 $\mathbf{u}$ 取负值，即使只有一个分量为负值，那么该步长也必须被分成几个小步长，有时由于步长尺寸的剧烈减小，求解速度极其缓慢。根本问题在于，共轭梯度方法对海赛矩阵的逆矩阵信息的收集是一点一点进行的，而且在搜索空间的位置也随之变动。当一个非线性函数与理论上的二次函数有相当大的区别时，旧信息在得到充分利用之前，就已变得毫无用处。

斯特林及其合作者[6,7,13,17]提出了一种复杂但却相当成功的方法。该方法在不断搜寻最小值的过程中，不再沿某个单一的搜索方向，而是在一个由向量张成的低（通常是3）维子空间中进行，这些向量在每点都要重新计算。子空间基向量的选取必须满足避开导致负值的方向。下面这个由向量张成的三维子空间就是最成功的选取方法之一，这些向量的分量由下式给出

$$e_\mu^{(1)} = \hat{u}_\mu[\nabla\mathscr{A}]_\mu$$

$$e_\mu^{(2)} = \hat{u}_\mu[\nabla\mathscr{B}]_\mu$$

$$e_\mu^{(3)} = \frac{\hat{u}_\mu\sum_\rho(\partial^2\mathscr{A}/\partial\hat{u}_\mu\,\partial\hat{u}_\rho)\hat{u}_\rho[\nabla\mathscr{B}]_\rho}{\sqrt{\sum_\rho\hat{u}_\rho([\nabla\mathscr{B}]_\rho)^2}} - \frac{\hat{u}_\mu\sum_\rho(\partial^2\mathscr{A}/\partial\hat{u}_\mu\,\partial\hat{u}_\rho)\hat{u}_\rho[\nabla\mathscr{A}]_\rho}{\sqrt{\sum_\rho\hat{u}_\rho([\nabla\mathscr{A}]_\rho)^2}} \tag{18.7.20}$$

（在这些方程中，不对 $\mu$ 求和。）如果认为点乘位于一个度量为 $g_{\mu\nu}=\delta_{\mu\nu}/u_\mu$ 的度量空间中，那么可以证明 $e^{(3)}$ 的选取将“远离”零值，见[6]。

在三维子空间中，三个分量的梯度与九个分量的海赛矩阵可以通过从大空间的投影来计算，子空间中的最小值可以根据（普通地）解三个联立线性方程组（如同第10.7节中的方程（10.7.4）一样）来估计。步长 $\Delta\mathbf{u}$ 的大小必须满足下述不等式

$$\sum_\mu(\nabla\hat{u}_\mu)^2/\hat{u}_\mu < (0.1\text{ 至 }0.5)U \tag{18.7.21}$$

由于梯度方向 $\nabla\mathscr{A}$ 和 $\nabla\mathscr{B}$ 单独使用，因此可以同时进行最小值的搜寻及 $\lambda$ 的调整，并使其最终满足预期的约束条件。这其中有多种技巧可以采用。

Cornwell 和 Evans 提出了一个虽不普遍，但在实际应用中却常常令人满意的方法[18]。应该注意，$\mathscr{B}$ 的海赛矩阵（二阶偏导数）是一个对角矩阵，那么 $\mathscr{A}$ 的海赛矩阵是否也存在一个有用的对角矩阵与它近似相等，例如 $2\mathbf{A}^T\cdot\mathbf{A}$；如果用 $\Lambda_\mu$ 表示该近似矩阵的对角元，那么 $\mathbf{u}$ 的一个很有用的步长应该是

$$\nabla \hat{u}_\mu = -\frac{1}{\Lambda_\mu + \lambda f''(\hat{u}_\mu)}(\nabla \mathscr{A} + \lambda \nabla \mathscr{B}) \qquad (18.7.22)$$

(与方程(10.7.1)比较。)更极端的情形是,找一个对角元素为常数的近似对角矩阵,$\Lambda_\mu = \Lambda$。于是

$$\nabla \hat{u}_\mu = -\frac{1}{\Lambda + \lambda f''(\hat{u}_\mu)}(\nabla \mathscr{A} + \lambda \nabla \mathscr{B}) \qquad (18.7.23)$$

由于 $\mathbf{A}^T \cdot \mathbf{A}$ 具有双卷积点分布函数的一些性质,以及由于实际情形中的点分布函数常常具有一个尖锐的中心峰值,因此这些极端情形也常常是富有成效的。

先根据 $A_{i\mu}$ 对 $\Lambda$ 做一个粗略的估计,例如,

$$\Lambda \sim \left(\sum_i [A_{i\mu}]^2\right) \qquad (18.7.24)$$

精确值并不重要,因为在实际情况下,$\Lambda$ 是要进行自适应调整的:如果 $\Lambda$ 太大,则方程(18.7.23)的步长将会太小(即同一方向上较大的步长将使 $\mathscr{A}+\lambda\mathscr{B}$ 减小得更多)。如果 $\Lambda$ 太小,则试验步长将位于一个不可能的区域($u_\mu$ 的负值),或者产生一个较大的 $\mathscr{A}-\lambda\mathscr{B}$。此处 $\Lambda$ 的调整与第15.5节中的勒温伯格-马阔特方法明显很相似;这是因为 MEM 与非线性最小二乘拟合问题极其相似,因此不应该觉很意外,参考文献[12]还讨论了如何利用 $\Lambda + \lambda f''(u_\mu)$ 值去调整拉格朗日乘子 $\lambda$,从而使问题收敛到 $\chi^2$ 的期望值。

所有实用的 MEM 算法都必须进行大约 30~50 次迭代后才能收敛。目前还没有什么方法可以使收敛问题得到根本解决。

### 18.7.2 "贝叶斯"与"历史性"的最大熵

最大熵图像恢复的一些最新进展都表明它们属于"贝叶斯",而区别于先前的"历史性"方法。详见参考文献[13]。

- 更好的先验:我们已经注意到,最大熵函数(方程(18.7.13))在量化所有象素时是不变的,而且也不涉及光滑性。所谓的"本质关联函数"(ICF)模型与熵函数十分类似,从而算法也很类似。但它要用到邻近关联象素的值,因此有光滑的作用。
- 更好的 $\lambda$ 估计值:在上文中,我们适当地选取 $\lambda$ 使 $\chi^2$ 落在预期的统计意义上的狭小区域 $N \pm (2N)^{1/2}$ 中。但这实际上是过大地估计了 $\chi^2$,因为在重建过程中,参数的一些有效数 $\gamma$ 是"固定"的。贝叶斯方法则能得到该 $\gamma$ 一个自相容的估计值,并使 $\lambda$ 的选取更客观。

参考文献和进一步读物:


Jaynes, E.T. 1976. in *Foundations of Probability Theory, Statistical Inference, and Statistical Theories of Science*, W.L. Harper and C.A Hooker, eds. (Dordrecht: Reidel). [1]

Jaynes, E.T. 1985. in *Maximum-Entropy and Bayesian Methods in Inverse Problems*, C.R. Smith and W.T.Grandy, Jr., eds. (Dordrecht: Reidel) [2]

Jaynes, E.T. 1984, in *SIAM-AMS Proceedings*, vol. 14, D.W. McLaughlin, ed. (Providence, RI: American Mathematical Society). [3]

Titterington, D.M. 1985, *Astronomy and Astrophysics*, vol. 144, 381~387. [4]

Narayan, R., and Nityananda, R. 1986, *Annual Review of Astronomy and Astrophysics*, vol. [illegible]

~170. [5]

Skilling, J., and Bryan, R.K. 1984, *Monthly Notices of the Royal Astronomical Society*, vol. 211, pp. 111~124. [6]

Burch, S.F., Gull, S.F., and Skilling, J. 1983, *Computer Vision, Graphics and Image Processing*, vol. 23, pp. 113~128. [7]

Skilling, J. 1989, in *Maximum Entropy and Bayesian Methods*, J. Skilling, ed. (Boston: Kluwer). [8]

Frieden, B.R. 1983, *Journal of the Optical Society of America*, vol. 73, pp. 927~938. [9]

Skilling, J., and Gull, S.F. 1985, in *Maximum-Entropy and Bayesian Methods in Inverse Problems*. C.R. Smith and W.T. Grandy, Jr., eds. (Dordrecht: Reidel). [10]

Skilling, J. 1986, in *Maximum Entropy and Bayesian Methods in Applied Statistics*, J.H. Justice, ed. (Cambridge: Cambridge University Press). [11]

Cornwell, T.J., and Evans, K.F. 1985, *Astronomy and Astrophysics*, vol. 143, pp. 77~83. [12]

Gull, S.F. 1989, in *Maximum Entropy and Bayesian Methods*, J. Skilling, ed. (Boston: Kluwer). [13]

# 第十九章 偏微分方程

## 19.0 引言

偏微分方程的数值解法就其自身来说，是一个广阔的研究课题。偏微分方程即使不是绝大多数，也是许多连续物理系统，如流体、电磁场及人体等的计算机分析或模拟的核心所在。本章旨在给出最简单明了的介绍。理想的情形，应该有一部完整的**数值解法**第二卷专门解决偏微分方程。（当然，参考文献[1~4]提供适用的参考价值。）

在大多数数学著作中，偏微分方程（PDEs）根据其**特征**或**信息传播**曲线而分为三种类型，即**双曲型、抛物型和椭圆型**。双曲型方程的典型例子是一维**波动**方程：

$$\frac{\partial^2 u}{\partial t^2} = v^2 \frac{\partial^2 u}{\partial x^2} \tag{19.0.1}$$

其中 $v$=常数，为波的传播速度。典型的抛物型方程是扩散方程：

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(D\frac{\partial u}{\partial x}\right) \tag{19.0.2}$$

其中 $D$ 是扩散系数。典型的椭圆型方程是泊松方程：

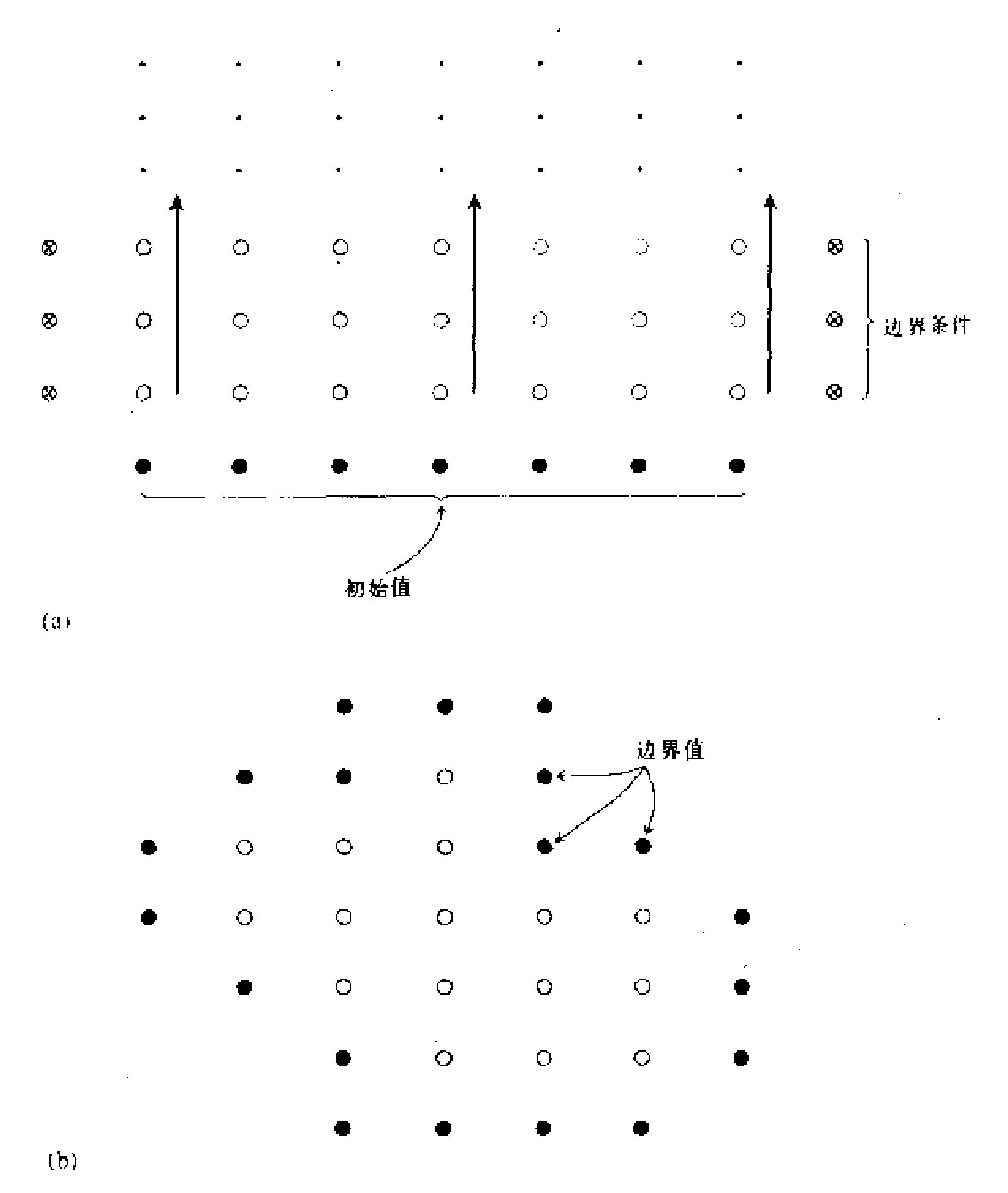
$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \rho(x,y) \tag{19.0.3}$$

其中源项 $\rho$ 给定。如果该源项等于零，则方程称为**拉普拉斯方程**。

从计算的角度来看，这种分为三种标准类型的分类并不是非常有意义的，至少不如一些其它的本质性区分来得重要。方程(19.0.1)和方程(19.0.2)都定义了**初值问题**或称为**柯西问题**：如果有关 $u$ 的信息（可能包括 $u$ 的时间导数信息）在某个初始时刻 $t_0$ 对任意 $x$ 是已知的，则方程描述了 $u(x,t)$ 如何随时间向前传播它自身。换言之，方程(19.0.1)和(19.0.2)描述了 $u$ 随时间的演变。一个数值解的目的在于：按某预定精确度，追踪这个随时间的演变过程。

相反的，方程(19.0.3)则要我们去找一个“静态”的函数 $u(x,y)$，使之在 $(x,y)$ 的讨论区域里满足该方程，并且——特别要指明的是——还必须使之在讨论的区域的边界上，具有某些期望的特性。这类问题被称作**边界值问题**。由于一个初值问题可“沿时间向前积分”，在这一相同意义下，一般边界值问题不可能是稳定地正好“从边界向里积分”。因此，数值解的目的是：以某种方式尽快在任意位置收敛到正确解。

因此，从计算的角度来看，最重要的分类方法是：眼前的问题是**初值**（随时间的发展）问题还是**边界值**（静态解）问题，图19.0.1形象表示了它们之间的区别。注意，虽然黑体字的术语是标准用语，但从计算的角度来看，括号中的术语却是对这两类问题的更好的描述。初值问题再分成双曲型和抛物型就不怎么重要了，因为(i)许多实际问题都是混和型的，(ii)正如我们将要看到的，在人们讨论实际的计算方案时，大多数双曲型问题都混杂有抛物型的成分。




在(a)中，初值被给定在一个"时间片"上，并且希望沿箭头所示方向逐行计算空心圆点的解，得到随时间向前发展的解。而在每一行左边和右边(⊗)的边界条件也必须满足，但是每一个时刻仅计算一行，在内存中，只需保留先前的一行或少数几行。在(b)中，边界值沿一个网格的周界给定，并使用一个迭代过程以确定所有内部点(空心点)的值。所有网格点都必须保留在内存中。

图19.0.1　初值问题(a)和边界值问题(b)的对比

## 19.0.1　初值问题

定义一个初值问题需要回答如下问题：

- 随时间向前传播的应变量是什么?
- 每个变量的发展方程是什么?通常这些发展方程都是耦合的，即在每个方程的右端，同时出现了不止一个的应变量。
- 每个变量的发展方程中，出现的最高阶时间导数是什么?如果可能，关于时间的导数应该单独放在方程的左端。不仅是变量的值，而且包括它的所有时间导数的值——直

至其最高阶导数——都必须确定，以定义这个发展方程。

- 约束着有关空间区域的边界点，随时间发展的特殊方程（即边界条件）是什么？例如，**狄利克莱**条件，把边界点的值定义为时间的函数；**诺伊曼**条件，则赋予了边界上的法向梯度值；**行波边界条件**，正是所说有条件。

本章19.1节～19.3节将介绍几种不同形式的初值问题。我们的内容并不完全，但力图通过一些仔细挑选的典型范例，给读者提供大量概括性的知识。这些例子说明一个重要的思想：在原则上，计算时须特别注意算法的稳定性。在初值问题中，许多貌似合理的算法，事实上并不能运行——其数值是不稳定的。

### 19.0.2 边界值问题

定义一个边界值问题需回答如下问题：

- 变量是什么？
- 哪些方程在讨论的区域内部成立？
- 哪些方程在讨论区域的边界上逐点成立？（对二阶椭圆方程，狄利克莱条件和诺伊曼条件是极可能的选择条件，但也可能遇到更加复杂的边界条件。）

与初值问题相反，对于边界值问题，稳定性相对容易获得。这样一来，算法的效率，包括计算量和存储的需求，就成了原则上的重点。

由于边界值问题中，所有的条件都必须"同时地"满足，因此这些问题至少在概念上，通常可简化为大量联立代数方程的求解。当这些方程为非线性时，通常采用线性化或迭代的办法来求解。因此，不失一般性，我们可以把边界值问题看求解特殊的大线性方程组。

作为将在第19.4节～第19.6节中作为"典型问题"引用的一个例子，让我们考虑一下方程（19.0.3）的**有限差分解法**。用函数 $u(x,y)$ 在离散点集上的取值来代表函数自身

$$\begin{aligned} x_j &= x_0 + j\Delta, \qquad j = 0,1,\ldots,J \\ y_l &= y_0 + l\Delta, \qquad l = 0,1,\ldots,L \end{aligned} \tag{19.0.4}$$

其中 $\Delta$ 为**网格间距**。从现在起，我们用 $u_{j,l}$ 表示 $u(x_j,y_l)$，用 $\rho_{j,l}$ 表示 $\rho(x_j,y_l)$。用一个有限差分表达式（见图19.0.2）代替方程（19.0.3）：

$$\frac{u_{j+1,l} - 2u_{j,l} + u_{j-1,l}}{\Delta^2} + \frac{u_{j,l+1} - 2u_{j,l} + u_{j,l-1}}{\Delta^2} = \rho_{j,l} \tag{19.0.5}$$

或者等价地为：

$$u_{j+1,l} + u_{j-1,l} + u_{j,l+1} + u_{j,l-1} - 4u_{j,l} = \Delta^2\rho_{j,l} \tag{19.0.6}$$

为了把这个线性方程组系统写为矩阵形式，我们需要从 $u$ 中构造一个向量。考虑用一个一维序列来标记网格点上的二维信息：

$$i \equiv j(L+1) + l \quad 对 \quad j = 0,1,\ldots,J, \quad l = 0,1,\ldots,L \tag{19.0.7}$$

换言之，沿表示 $y$ 值的列数，$i$ 上升得最快。方程（19.0.6）现在在为：

$$u_{i+L+1} + u_{i-(L+1)} + u_{i+1} + u_{i-1} - 4u_i = \Delta^2\rho_i \tag{19.0.8}$$

该方程仅在内点 $j=1,2,\cdots J-1$；$l=1,2,\cdots L-1$ 处成立。

位于

$j=0$ [即 $i=0,\ldots,L$]

$j=J$ [即 $i=J(L+1),\ldots,J(L+1)+L$]

$$
\begin{aligned}
l &= 0 \quad [\text{即 } i = 0, L+1, \ldots, J(L+1)] \\
l &= L \quad [\text{即 } i = L, L+1+L, \ldots, J(L+1)+L]
\end{aligned} \tag{19.0.9}
$$

处的点为边界点。在这些点上，或者是 $u$，或者是 $u$ 的导数已经给定，如果我们把所有的"已知"信息都放到方程(19.0.8)的右端，则该方程具有形式为

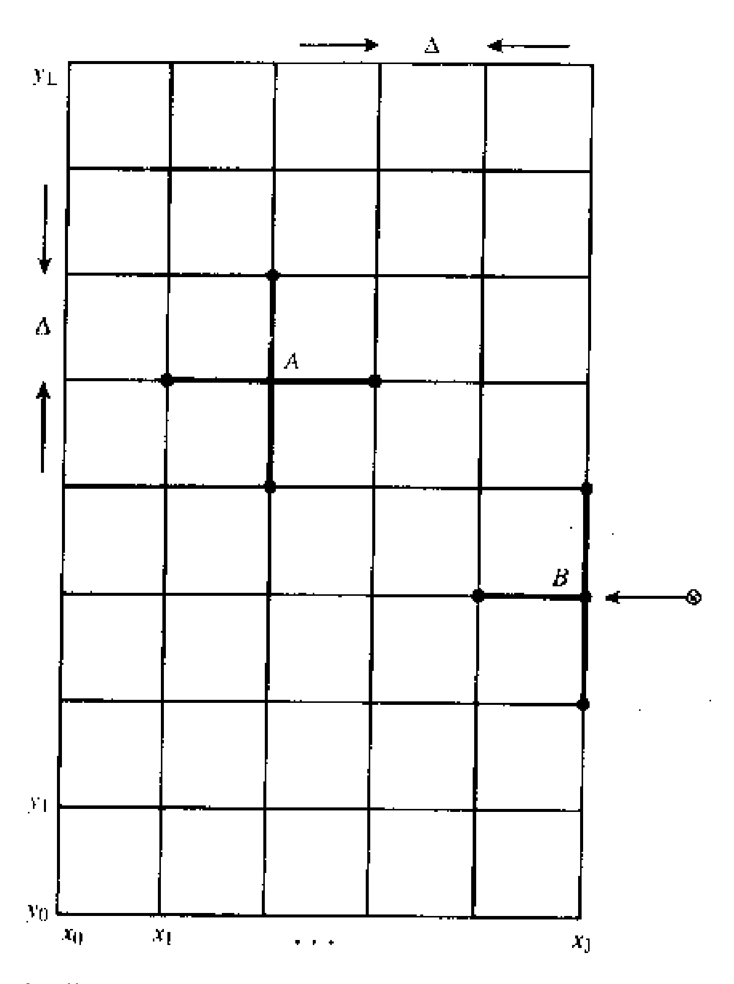
$$
\mathbf{A} \cdot \mathbf{u} = \mathbf{b} \tag{19.0.10}
$$

其中矩阵 **A** 具有如图19.0.3所示的形式。

矩阵 **A** 的被称为"带边的三对角型"矩阵，一个常见的二阶线性椭圆方程：

$$
\begin{aligned}
a(x,y)\frac{\partial^2 u}{\partial x^2} + b(x,y)\frac{\partial u}{\partial x} + c(x,y)\frac{\partial^2 u}{\partial y^2} - d(x,y)\frac{\partial u}{\partial y} \\
+ e(x,y)\frac{\partial^2 u}{\partial x \partial y} - f(x,y)u = g(x,y)
\end{aligned} \tag{19.0.11}
$$

其相应矩阵除了非零项不为常数外，有着与 **A** 相似的结构。




在 $A$ 点的二阶导数用图中所示的与 $A$ 相连接的点来估算；在 $B$ 点的二阶导数也用相连接点来估算，同时还用到"右端项"的边界信息，用符号⊗表示。

图19.0.2　在二维网格上，二阶椭圆方程的有限差分图

粗略地分类，有三种不同的方法可求解方程(9.0.10)，当然并非处处适用：松驰法、"快速"法(如傅里叶方法)和直接矩阵法。

松驰法直接利用稀疏矩阵 **A** 的结构，把矩阵 **A** 分成两部分：

$$\mathbf{A} = \mathbf{E} - \mathbf{F} \tag{19.0.12}$$

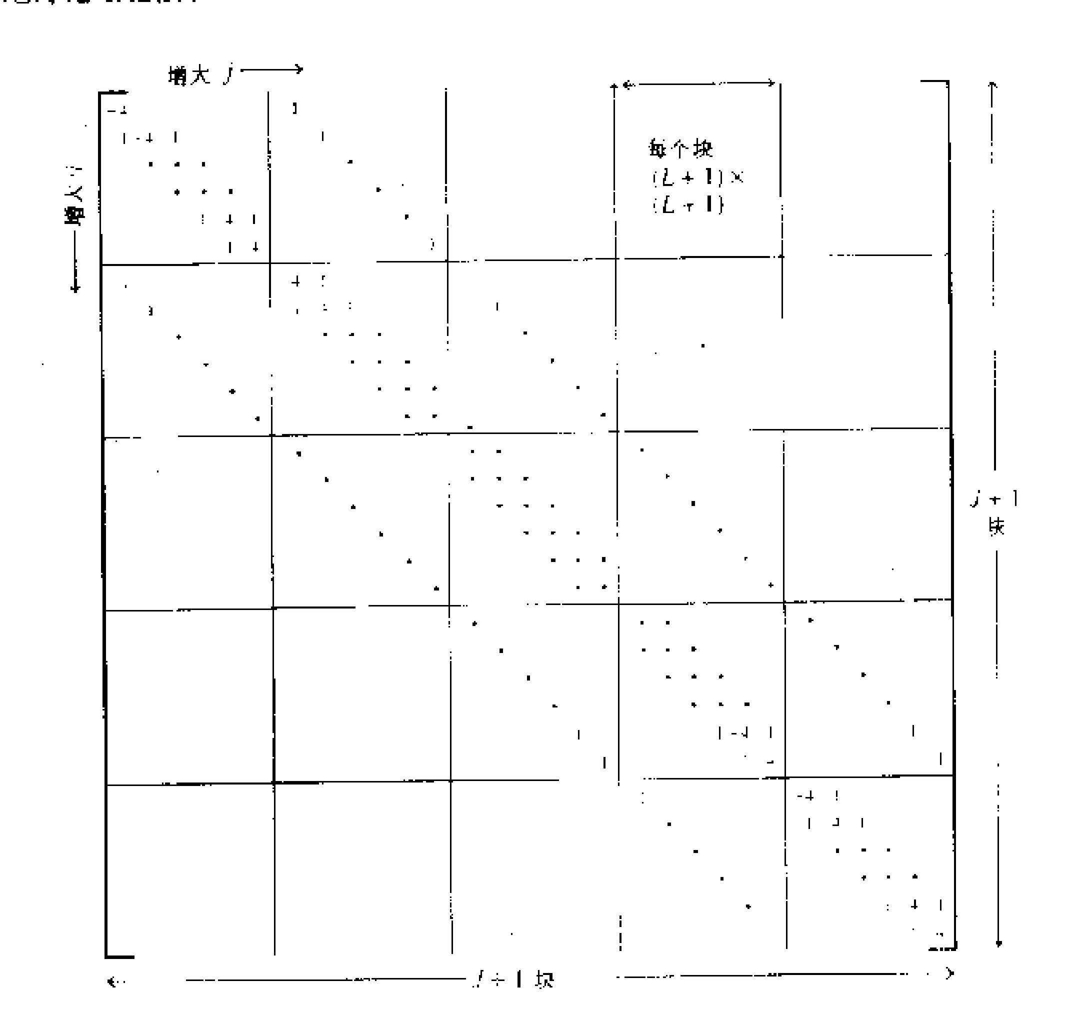
其中，$\mathbf{E}$ 是容易求逆的矩阵，$\mathbf{F}$ 是余值。因而式(19.0.10)变成：

$$\mathbf{E} \cdot \mathbf{u} = \mathbf{F} \cdot \mathbf{u} + \mathbf{b} \tag{19.0.13}$$

松驰法包括选择 $\mathbf{u}$ 的初始值 $\mathbf{u}^{(0)}$，然后通过逐次求解，从

$$\mathbf{E} \cdot \mathbf{u}^{(r)} = \mathbf{F} \cdot \mathbf{u}^{(r-1)} + \mathbf{b} \tag{19.0.14}$$

迭代获得 $\mathbf{u}^{(r)}$。由于选择 $\mathbf{E}$ 容易求逆，因而每步迭代都很迅速。我们将在19.5节和19.6节较详细地讨论松驰法。




所有未标明的元素为0，该矩阵有自身为三对角线型的对角块，这种形式的矩阵被称为"带边的三对角型矩阵"，一个如此稀疏的矩阵不必按图示的完整的形式存储。

图19.0.3　由二阶椭圆方程(在此为方程19.0.6)导出的矩阵结构

所谓的快速法[5]仅适用一类相当特殊的方程：这些方程具有常系数，或者更一般地，在已选择的坐标系中，它们是可分离的；另外，边界必须与坐标轴重合。这类特殊方程在实际应用中经常遇到，我们放到19.4节做详细讨论。但请注意，将在19.6节讨论的多网格松驰法可能比"快速法"还要快。

矩阵法力图直接求解方程

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{19.0.15}$$

其实用程度极强地依赖于当前问题中矩阵 A 的具体结构，因此，我们在这一点上的讨论，仅限于一些评述和提示。

矩阵的稀疏程度必须首先考虑，否则矩阵问题将非常巨大。例如，一个在 100×100 的空间网络上最简单的问题，将涉及到 10000 个未知的 $u_{j,l}$，它蕴含一个 10000×10000 的矩阵 A，包含 $10^8$ 个元素！

正如我们在第2.7节末尾讨论的，如果 A 是对称正定的（在椭圆问题中通常如此），那么可以使用共轭梯度算法。在实际应用中，舍入误差经常破坏了共轭梯度算法求解有限差分方程的效果。但是在这种方法中，如果首先改写方程，使矩阵 A 变换成与单位阵近似的矩阵 A′，那么上方法还是可用的。此时，这些方程定义的二次曲面都具有近似的球形轮廓，并且共轭梯度算法运行效果很好。在第2.7节的程序 **linbcg** 中，一个类似的预条件器被用来以更一般的双共轭梯度法求解非正定问题。至于在 PDEs 中出现的正定情形，一个成功的使用此方法的例子是 **Cholesky 的不完全共轭梯度法**（ICCG）（参见[6~8]）。

另一种依赖于变换手段的方法是斯通（Stone）提供的**强隐式过程**[9]，一个运用该过程名为 SIPSOL 的程序已发表，参见[10]。

第三类矩阵法是如第2.7节所描述的分析——因子分解——运算的方法。

总而言之，当所具有的内存可以使用上述矩阵法——通常是不超过 $10^6$，但却比松弛法所需容量大得多时——则可以考虑使用之。此时只有多网格松驰法（第19.6节）可与最佳的矩阵法相媲美。但当网格大于300×300，我们一般会发现，只有松驰法，或"快速"法（在其可用时），才是可能的方法。

## 19.0.3 有限差分以外的众多方法

除了有限差分方法，还有一些方法可以求解偏微分方程，最重要的有有限元法、蒙特卡罗法、谱方法和变分法。遗憾的是，我们在本章只能对有限差分法作一些仔细介绍，本书将不对其它方法进行讨论。有限元法[11~12]尤为固体机械学和结构工程学的实际工作者们偏爱，这种方法在设置计算元素的位置时，提供了相当大的自由度，在处理高度不规则的地学问题时，也非常重要；谱方法[13~15]适于非常规则的地学问题和光滑函数，其收敛比有限差分法快得多（参看第19.4节），但对不连续的问题，该方法效果并不好。

参考文献和进一步读物：


Ames，W. F. 1977，*Numerical Methods for Partial Differential Equations*，2nd ed.（New York：Academic Press）. [1]

Richtmyer，R. D.，and Morton，K. W. 1967，*Difference Methods for Initial Value Problems*，2nd ed.（New York：Wiley-Interscience）. [2]

Roache，P. J. 1976，*Computational Fluid Dynamics*（Albuquerque：Hermosa）. [3]

Mitchell，A. R.，and Griffiths，D. F. 1980，*The Finite Difference Method in Partial Differential Equations*（New York：Wiley）[includes discussion of finite element methods]. [4]

Dorr，F. W. 1970，*SIAM Review*，vol. 12，pp. 248~263. [5]

Meijerink，J. A.，and van der Vorst，H. A. 1977，*Mathematics of Computation*，vol. 31，pp. 148~162. [6]

Van der Vorst，H. A. 1981，and *Journal of Computational Physics*，vol. 44，pp. 1-19 [review of sparse it

erative methods]. [7]

Kershaw, D. S. 1970, *Journal of Computational Physics*, vol. 26, pp. 43~65. [8]

Stone, H. J. 1968, *SIAM Journal on Numerical Physics*, vol. 5, pp. 530~558. [9]

Jesshope, C. R. 1979, *Computer Physics Communications*, vol. 17, pp. 383~391. [10]

Strang, G., and Fix, G. 1973, *An Analysis of the Finite Element Method* (Englewood Cliffs, NJ: Prentice-Hall). [11]

Burnett, D. S. 1987, *Finite Element Analysis: From Concepts to Applications* (Reading, MA: Addison Wesley). [12]

Gottlieb, D. and Orszag, S. A. 1977, *Numerical Analysis of Spectral Methods: Theory and Applications* (Philadelphisa: S. I. A. M. ). [13]

Canuto, C., Hussaini, M. Y., Quarteroni, A., and Zang, T. A. 1988, *Spectral Methods in Fluid Dynamics* (New York: Springer-Verlag). [14]

Boyd, J. P. 1989, *Chebyshev and Fourier Spectral Methods* (New York: Springer-Verlag). [15]

## 19.1 通量守恒的初值问题

一维空间的一大类偏微分方程初值(随时间发展)问题,可以归入**通量守恒方程**的形式:

$$\frac{\partial \mathbf{u}}{\partial t} = -\frac{\partial \mathbf{F}(\mathbf{u})}{\partial x} \tag{19.1.1}$$

其中,u 和 F 为向量,且(在某些情况下)F 不但依赖于 u,而且依赖于 u 的空间导数,称向量 **F** 为**守恒通量**。

例如,典型的双曲型方程,具有常数传播速度 $v$ 的一维波动方程:

$$\frac{\partial^2 u}{\partial t^2} = v^2 \frac{\partial^2 u}{\partial x^2} \tag{19.1.2}$$

上式可以改写成两个一阶方程的方程组:

$$\begin{aligned} \frac{\partial r}{\partial t} &= v\frac{\partial s}{\partial x} \\ \frac{\partial s}{\partial t} &= v\frac{\partial r}{\partial x} \end{aligned} \tag{19.1.3}$$

其中

$$\begin{aligned} r &\equiv v\frac{\partial u}{\partial x} \\ s &\equiv \frac{\partial u}{\partial t} \end{aligned} \tag{19.1.4}$$

在这种情况下,$r$ 和 $s$ 成为 u 的两个分量,而通量 F 则由线性矩阵关系确定:

$$\mathbf{F}(\mathbf{u}) = \begin{pmatrix} 0 & -v \\ -v & 0 \end{pmatrix} \cdot \mathbf{u} \tag{19.1.5}$$

(精通物理学的读者可能会发现,方程(19.1.3)类似于麦克斯韦(Maxwell)的一维电磁波传播方程。)

在本节,我们将考虑一般的通量守恒方程(19.1.1)的一个典型例子,即关于 $u$ 为标量的方程:

$$\frac{\partial u}{\partial t} = -v\frac{\partial u}{\partial x} \tag{19.1.6}$$

其中 $v$ 是常数。对这种情形，我们已经定性地知道，该方程的通解是一列沿 x 轴正方向传播的波：

$$u = f(x - vt) \tag{19.1.7}$$

其中 $f$ 为一任意的函数。当然，我们探讨的数值解法将同样适用于式(19.1.1)所表示的更加一般化的方程。在一些文章中，方程(19.1.6)被称为**对流方程**，因为量 $u$ 是通过速度为 $v$ 的"流体流动"来传送的。

如何对方程(19.1.6)(或类似地，对式(19.1.1))进行有限差分呢？一种简单明了的方法是，沿 $t$ 轴和 $x$ 轴作等距划分，等分点可表示为：

$$\begin{aligned} x_j &= x_0 + j\Delta x \qquad j = 0,1,\ldots,J \\ t_n &= t_0 + n\Delta t \qquad n = 0,1,\ldots,N \end{aligned} \tag{19.1.8}$$

用 $u_j^n$ 表示 $u(t_n,x_j)$，我们有几种方法表示 $u$ 的时间导数项，一种明显的方法是取

$$\left.\frac{\partial u}{\partial t}\right|_{j,n} = \frac{u_j^{n+1} - u_j^n}{\Delta t} + O(\Delta t) \tag{19.1.9}$$

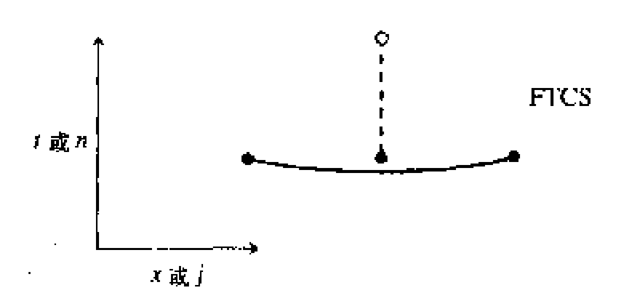
这叫做**欧拉向前差分法**(参见方程(16.1.1))。虽然欧拉向前差分法只有一阶精度，但它有一个优点，就是仅用第 $n$ 时间层的已知值，即可计算 $n+1$时间层的值。对于空间导数，我们可以使用二阶精确的表示式，仍然只用到 $n$ 时间层的已知值：

$$\left.\frac{\partial u}{\partial x}\right|_{j,n} = \frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x} + O(\Delta x^2) \tag{19.1.9}$$

由此而得到的方程(19.1.5)的有限差分逼近称为 FTCS 表示(即时间向前空间中心差分)：

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = -v\left(\frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x}\right) \tag{19.1.11}$$

很容易将上式变换成利用其它量计算 $u_j^{n+1}$ 的公式。FTCS 格式的图解说明见图19.1.1。它是算法的一个很好的例子，易于推导，占据极少内存且计算迅速，但实际上却糟得无法运行。(请看下文。)




在该图和后面的图中，空心圆圈是希望求解所在的新点；实心圆圈是已知点，其函数值用于计算新点；实线连接用于计算空间导数的点；虚线则连接用于计算时间导数的点，对双曲型问题 FTCS 格式，通常不稳定，故通常不能使用。

图19.1.1 时间向前空间中心差分格式(FTCS)的图示

FTCS 表示是一种**显式格式**，这意味着任意 $j$ 值的 $u_j^{n+1}$ 均可由已知的值明确地算出。后面我们将遇到**隐式格式**，它要求我们对不同的 $j$ 求解耦合 $u_j^{n+1}$ 的隐式方程。(对于常微分方

程的显式和隐式方法在第16.6节已讨论了。)FTCS 算法也是一个**单层**格式的例子，因为为计算第 $n+1$ 时层的值，只需存储第 $n$ 时间层的值。

### 19.1.1 冯·诺伊曼稳定性分析

非常遗憾，方程(19.1.11)的用处很有限，它是一种**不稳定**的方法，只能用来(如果确实要用)研究一小段振荡时期里的波。为了寻找具有更普遍适用性的替代方法，我们必须先介绍一下**冯·诺伊曼稳定性分析**。

冯·诺伊曼(Von Neumann)分析是局部的：我们假设差分方程的系数变化非常缓慢，以致于可示为时间域和空间域中的常数。在此情况下，该差分方程的无关解或称为**特征模**，一律具有形式为

$$u_j^n = \xi^n e^{ikj\Delta x} \tag{19.1.12}$$

其中 $k$ 是实的空间波数(可取任意值)，$\xi=\xi(k)$ 是依赖于 $k$ 的复数。关键之处是，每个特征模的时间依赖性只不过是复数 $\xi$ 的连续整数次幂，因此，如果对**某些** $k$ 值，$|\xi(k)|>1$，则差分方程是不稳定的(即有指数上升的趋势)。$\xi$ 的值被称为在已知波数 $k$ 处的**放大因子**。

为了获得 $\xi(k)$，我们只需简单地将式(19.1.12)代入(19.1.11)，除以 $\xi^n$，得：

$$\xi(k) = 1 - i\frac{v\Delta t}{\Delta x}\sin k\Delta x \tag{19.1.13}$$

对任何 $k$ 值，它的模数均>1，因此，FTCS 格式是无条件不稳定的。

如果速度 $v$ 是 $t$ 和 $x$ 的函数，则在方程(19.1.11)中，我们应写入 $v_j^n$。在冯·诺伊曼稳定性分析中，我们仍将 $v$ 表示为常数，因为对 $v$ 缓慢变化的情形，分析是局部进行的。事实上，即使在 $v$ 为严格常数时，冯·诺伊曼分析也不能严格地处理在 $j=0$ 和 $j=J$ 处的终端效应。

更一般地，如果方程右端对 $u$ 是非线性的，那么通过改写 $u=u_0+\delta u$，将方程展开为 $\delta u$ 的线性方程，则冯·诺伊曼分析将可以线性化。假设 $u_0$ 的值已完全满足差分方程，则冯·诺伊曼分析旨在寻找 $\delta u$ 的一个不稳定特征模。

尽管冯·诺伊曼方法不很严密，但它一般得出正确的答案。而且比其它更仔细的方法容易使用得多，因此，我们只采用这种方法进行分析。(参见[1]，对其它稳定性分析性法的讨论。)

### 19.1.2 Lax 方法

FTCS 方法的不稳定性可以通过 Lax 的一个简单变换得到修正。替换时间导数项中的 $u_j^n$ 为其平均值(见图19.1.2)：

$$u_j^n \to \frac{1}{2}(u_{j+1}^n + u_{j-1}^n) \tag{19.1.14}$$

则式(19.1.11)变为：

$$u_j^{n+1} = \frac{1}{2}(u_{j+1}^n + u_{j-1}^n) - \frac{v\Delta t}{2\Delta x}(u_{j+1}^n - u_{j-1}^n) \tag{19.1.15}$$

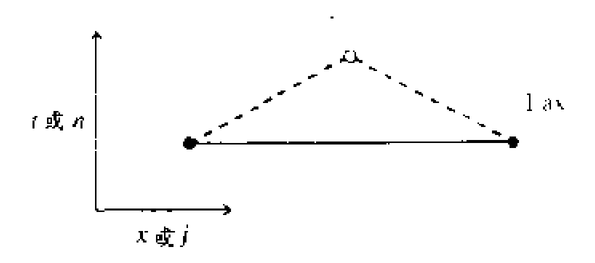
将之代入方程(19.1.12)中，我们发现放大因子：

$$\xi = \cos k\Delta x - i\frac{v\Delta t}{\Delta x}\sin k\Delta x \tag{19.1.16}$$

由稳定性条件 $|\xi|^2\leqslant 1$ 得要求：

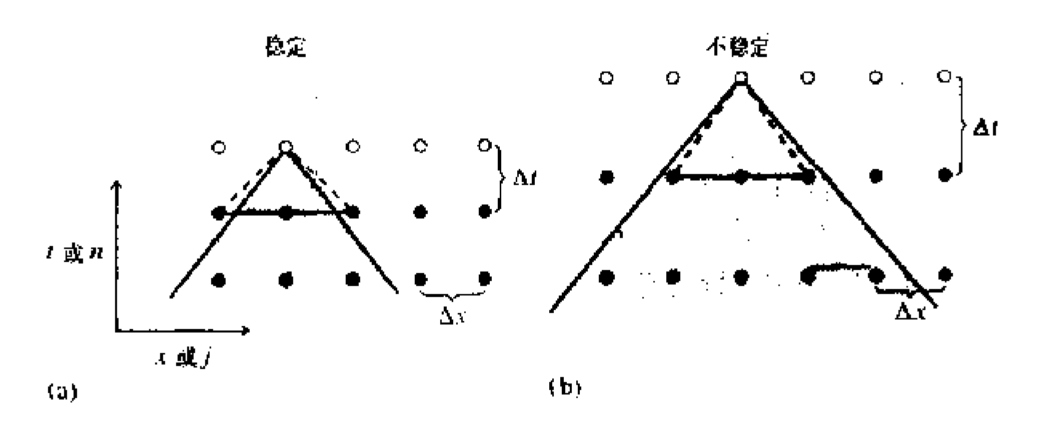
$$\frac{|v|\Delta t}{\Delta x} \leq 1 \tag{19.1.17}$$

这便是著名的 Courant-Friedriche-Lewy 稳定性准则，通常被简称为 **Courant 条件**。直观地，这种稳定性条件可以理解如下(见图19.1.3)：在方程(19.1.15)中，量 $u_j^{n+1}$ 由 $n$ 时刻在点 $j-1$ 和 $j+1$ 处的信息计算而得，换言之，$x_{j-1}$ 和 $x_{j+1}$ 是允许为 $u_j^{n+1}$ 传递信息的空间区域的边界。现在想一想在连续的波动方程中，信息以最大速度 $v$ 传播，如果点 $u_j^{n+1}$ 在图19.1.3)所示的阴影之外，则它需要从比该差分格式允许的位置更远处的点中获得信息，缺少的那些信息必导致不稳定。因此，$\Delta t$ 不能选得太大。




定义如前图，该格式的稳定性差别准则是 Courant 条件。

图19.1.2　Lax 差分格式的图示




一个初值问题的偏微分方程蕴含着：u 在某一点的值取决于过去的某依赖区域中的信息，在此示为阴影；一种差分格式也有其自身的依赖区域，该依赖区域由一个时间片上(以实心圆的连接表明)点的选取来确定，而时间片上的值用于确定一个新点(以虚线连接表明)。如果差分格式的依赖区域大于偏微分方程的依赖区域，则该差分方案是 Courant 稳定的，如图中(a)情形；而如果两者关系相反，则该差分格式不稳定，如图中(b)所示。

图19.1.3　差分格式稳定性的 Courant 条件

方程(19.1.14)的简单替换稳定了 FTCS 格式，这个令人吃惊的结果使我们第一次亲身体会到，偏微分方程的差分是一门科学，也是一种技巧。为了一定程度地减少这种技巧的神秘感，让我们改写方程(19.1.15)，使之具有方程(19.1.11)带有余项的形式，然后比较一下 FTCS 格式和 Lax 格式：

$$\frac{u_j^{n+1}-u_j^n}{\Delta t} = -v\left(\frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x}\right) + \frac{1}{2}\left(\frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{\Delta t}\right) \tag{19.1.18}$$

这恰恰就是方程：

$$\frac{\partial u}{\partial t} = -v\frac{\partial u}{\partial x} + \frac{(\Delta x)^2}{2\Delta t}\nabla^2 u \tag{19.1.19}$$

的 FTCS 表示。其中，一维情形下，$\nabla^2 = \partial^2/\partial x^2$。事实上，我们是在方程中加入了一个扩散项，或者，回忆一下粘滞流体运动的 Navier－Stokes 方程，是加了一个耗散项。因此 Lax 格式被认为具有**数值耗散**或**数值粘度**。我们在放大因子中也会发现这一点：除非 $|v|\Delta t$ 恰巧等于 $\Delta x$，否则有 $|\xi|<1$ 并且波的振幅假衰减。

假衰减是不是比假上升好些呢？回答是肯定的。我们希望精确研究的尺度是包含许多网格点，因此有 $k\Delta x<<1$（空间波数 k 在方程(19.1.12)中已定义）。对于这些尺度，放大因子在稳定格式和不稳定格式中，都可视为非常接近于1，因此稳定格式和不稳定格式基本上同等地精确。但是，对我们不感兴趣的短尺度，其中 $k\Delta x\sim 1$，使用不稳定格式却将摧毁和淹没解中有意义的部分。而稳定格式的情形就好得多，其中短波长部分毫无影响地消失。稳定格式和不稳定格式对这些短波长部分都是不精确的，但当格式稳定时，这种不精确是一种可以容忍的性质。

当独立变量 **u** 是一个向量时，冯·诺伊曼分析稍稍复杂。例如，我们可以考虑把方程(19.1.3)改写为：

$$\frac{\partial}{\partial t}\begin{bmatrix} r \\ s \end{bmatrix} = \frac{\partial}{\partial x}\begin{bmatrix} vs \\ vr \end{bmatrix} \tag{19.1.20}$$

对该方程使用 Lax 方法：

$$\begin{aligned} r_j^{n+1} &= \frac{1}{2}(r_{j+1}^n + r_{j-1}^n) + \frac{v\Delta t}{2\Delta x}(s_{j+1}^n - s_{j-1}^n) \\ s_j^{n+1} &= \frac{1}{2}(s_{j+1}^n + s_{j-1}^n) - \frac{v\Delta t}{2\Delta x}(r_{j+1}^n - r_{j-1}^n) \end{aligned} \tag{19.1.21}$$

通过假设特征模具在如下(向量)形式，继续进行冯·诺伊曼稳定性分析：

$$\begin{bmatrix} r_j^n \\ s_j^n \end{bmatrix} = \xi^n e^{ikj\Delta x}\begin{bmatrix} r^0 \\ s^0 \end{bmatrix} \tag{19.1.22}$$

其中，右端的向量(在空间域和时间域)是常特征向量，$\xi$ 和以前一样是复数。把式(19.1.22)代入(19.1.21)再除以幂 $\xi^n$，得齐次向量方程：

$$\begin{bmatrix} (\cos k\Delta x) - \xi & i\frac{v\Delta t}{\Delta x}\sin k\Delta x \\ i\frac{v\Delta t}{\Delta x}\sin k\Delta x & (\cos k\Delta x) - \xi \end{bmatrix} \cdot \begin{bmatrix} r^0 \\ s^0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{19.1.23}$$

当且仅当左边矩阵的行列式等于0时，该方程有解；由此条件，很容易得到 $\xi$ 的两个根：

$$\xi = \cos k\Delta x \pm i\frac{v\Delta t}{\Delta x}\sin k\Delta x \tag{19.1.24}$$

稳定的条件是两个根都满足 $|\xi|\leqslant 1$，结果恰好又是式(19.1.17)的 Courant 条件。

### 19.1.3 其它种类的误差

迄今为止，我们已经详细探讨了**振幅误差**，它与差分格式的稳定性和不稳定性有着紧密联系。当我们转移注意力到精确度上而不是稳定性上时，还有其它种类的误差与之相应。

双曲型方程的有限差分格式可以呈现色散或者**相位误差**。例如方程(19.1.16)可以重写

波：

$$\xi = e^{-ik\Delta x} + i\left(1 - \frac{v\Delta t}{\Delta x}\right)\sin k\Delta x \tag{19.1.25}$$

任意一个初始波群都具有不同 $k$ 值的波形的迭加。在每个时间层上、波形依据不同的 $k$ 值乘上一个不同的相位因子(19.1.25)。如果 $\Delta t = \Delta x/v$，则因为每个波形乘以 $\exp(-ik\Delta x)$，所以能得波群中每个波形的精确解 $f(x-vt)$。对于这个 $\Delta t$ 值，方程(19.1.25)表明有限差分方程解是精确的解析解。然而，若 $v\Delta t/\Delta x$ 不确切地等于1，波形的相位关系可能变得混乱不堪，并且波群发散。由式(19.1.25)可知，只要波长变得与网格间隔 $\Delta x$ 可相比拟，这种色散将变大。

第三类误差与非线性双典型方程相联系，因此有时被叫做**非线性不稳定性**。例如，流体运动的欧拉方程或 Navier-Stokes 方程的一部分可表示为：

$$\frac{\partial v}{\partial t} = - v\frac{\partial v}{\partial x} + \cdots \tag{19.1.26}$$

关于 $v$ 的非线性项可能导致傅里叶空间中，从长波长向短波长的能量转换，这引起波形变陡，直至垂直形状或"冲击波"发生。冯·诺伊曼分析中指出，稳定性取决于 $k\Delta x$，因此一种在平缓波形下稳定的格式，可能在陡峭波形下变得不稳定。当一种差分格式中，傅里叶空间的级联终止在网格上可代表的最短波长处，即 $k \sim 1/\Delta x$ 处时，这类麻烦将会出现。如果能量简单以这些方式累加，最终必然淹没我们感兴趣的长波长形式的能量。

非线性不稳定性和激波的形成，可一定程度地，通过前面讨论的方程(19.1.18)中的数值粘度来控制。但一些流体问题中，激波的形成不仅仅是件烦恼的事情，而且是一种其细节有待研究的、流体的实际物理行为，因此，仅仅有数值粘度尚不足够或尚不足以进行控制。我们将在后面从流体动力学角度对这个复杂的问题作进一步的讨论。

对于波动方程，传播误差(振幅或相位)经常是最令人头痛的，而对于对流方程，**传导误差**则往往更令人关注。在方程(19.1.15)的 Lax 格式中，对对流量 $u$ 在网格点 $j$ 处的扰动，将传至下一个时间层的网格点 $j-1$ 和 $j+1$。不过在实际中，如果速度 $v$ 是正的，则仅仅网格点 $j+1$ 会受到影响。

使传导性质得到"改善"的最简单方法是使用**迎风差分**法(见图19.1.4)：

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = - v_j^n\begin{cases}\dfrac{u_j^n - u_{j-1}^n}{\Delta x}, & v_j^n > 0 \\[2ex] \dfrac{u_{j+1}^n - u_j^n}{\Delta x}, & v_j^n < 0\end{cases} \tag{19.1.27}$$

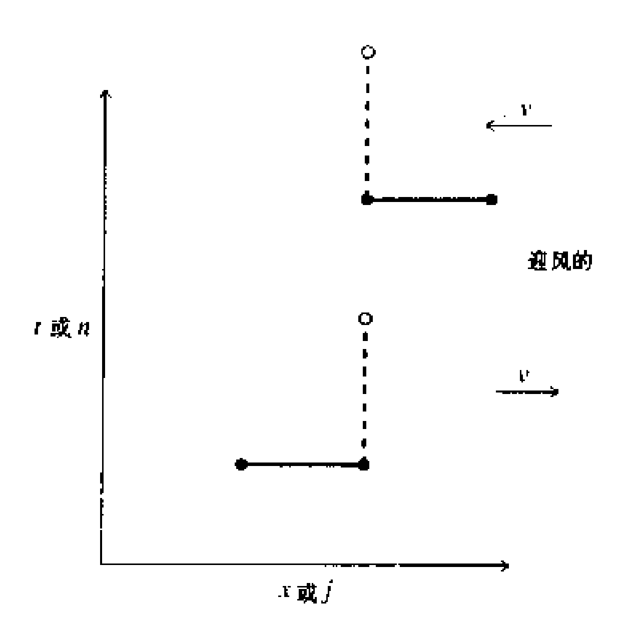
注意，这种格式在空间导数的计算上仅是一阶精确度，而不是二阶精确度。怎样才能使它"更好"呢？答案令数学家们不太满意：按严格的数学意义，数字模拟的目标并不一定是"精确"的，但在较不严格，更着重实效的意义下，有些时候，却能忠实于潜在的物理过程。由于这种原因，某些误差就比其它情形要容易接受得多。对于那些对流变量易于经历状态突变的问题，例如经历激波和其它不连续情况那样，迎风差分在总体上提高了问题的逼真性。对这类问题，读者就得根据自己问题的特殊物理性质来具体分析解决。

对于式(19.1.27)的差分格式，放大因子(对常值 $v$)是：

$$\xi = 1 - \left|\frac{v\Delta t}{\Delta x}\right|(1 - \cos k\Delta x) - i\frac{v\Delta t}{\Delta x}\sin k\Delta x \tag{19.1.28}$$

$$|\xi|^2 = 1 - 2\left|\frac{v\Delta t}{\Delta x}\right|\left(1 - \left|\frac{v\Delta t}{\Delta x}\right|\right)(1 - \cos k\Delta x) \tag{19.1.29}$$

由此可知，稳定性准则$|\xi|^2 \leqslant 1$恰好又是Courant条件(19.1.27)。




当对流常数$v$是负的时，如图中所示，上面的格式是稳定的；而当对流常数$v$是正的时，也正如图中所示，下面的格式是稳定的，当然，Courant条件也是必须要满足的。

图19.1.4 迎风差分格式的图示

有多种方法可改进一阶迎风差分的精确度。在连续性方程中，起初相距$v\Delta t$远的物质在一个$\Delta t$的时间间隔后，到达预定的点。在一阶方法中，物质通常从相距$\Delta x$以远的地方而来，如果$v\Delta t << \Delta x$(以确保精确度)，可能会导致大的误差。一种减少这种误差的方法是，在传导之前，$j-1$和$j$之间对$u$进行插值，这就得到了一种有效的二阶方法。二阶迎风差分的各种格式在[2~3]中进行了讨论和比较。

## 19.1.4 时间域上的二阶精确度

在使用一个时间域中一阶精确，空间域中二阶精确的方法时，人们往往不得不使$v\Delta t$远远小于$\Delta x$，例如至少有$5v\Delta t \leqslant \Delta x$，以获得期望的精确度。因而，在实际运用中，Courant条件实质并不是这些格式的限制因素。好在有在时间域中和空间域中均为二阶精确的格式，并且，这些格式通常可与它们的稳定性限制相吻合，故而只用到相对较少的计算次数。

例如，对于守恒方程(19.1.1)的**交错跳步**方法可以定义如下(见图19.1.5)：利用在时间$t^n$时的值$u^n$，计算流量$F_j^n$，然后利用流量的时间中心值，计算新值$u^{n+1}$：

$$u_j^{n+1} - u_j^{n-1} = -\frac{\Delta t}{\Delta x}(F_{j+1}^n - F_{j-1}^n) \tag{19.1.30}$$

交错跳步的名称取自这样一个事实：时间导数项中的时间层按"跳步"跃过空间导数项中的时间层。这种方法需要存储$u^{n-1}$与$u^n$以计算$u^{n+1}$。

对于我们简单的典型方程(19.1.6)，交错跳步法采用格式：

$$u_j^{n+1} - u_j^{n-1} = -\frac{v\Delta t}{\Delta x}(u_{j+1}^n - u_{j-1}^n) \tag{19.1.31}$$

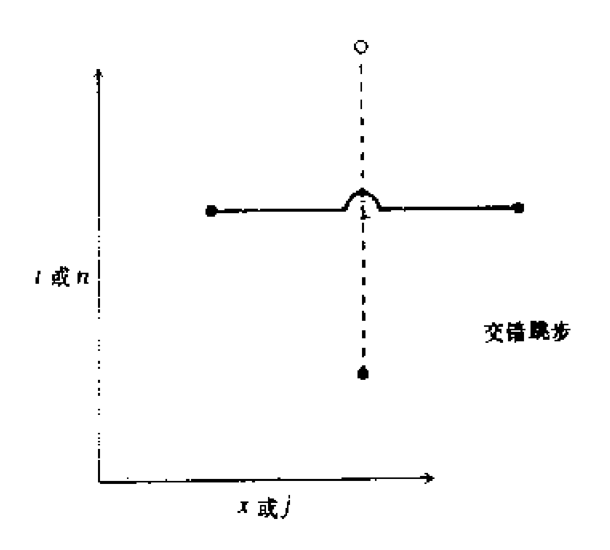
由于把特征模表达式(19.1.12)代入方程(19.1.31)，出现了 $\xi$ 的三个相邻次幂，故此时，冯·诺伊曼稳定性分析给出了 $\xi$ 的一个二次方程，而不是线性一次方程，

$$\xi^2 - 1 = -2i\xi\frac{v\Delta t}{\Delta x}\sin k\Delta x \tag{19.1.32}$$

其解为：

$$\xi = -i\frac{v\Delta t}{\Delta x}\sin k\Delta x \pm \sqrt{1 - \left(\frac{v\Delta t}{\Delta x}\sin k\Delta x\right)^2} \tag{19.1.33}$$

可见 Courant 条件再次成为稳定的条件。事实上，在方程(19.1.33)中，对任意 $v\Delta t \leqslant \Delta x$，均有 $|\xi|^2 = 1$。这正是交错跳步法的巨大优势：没有振幅的发散。




注意，来自先前两个时间层的信息被用于获得期望点，这种格式在时间和空间中均是二阶精确的。

图19.1.5　交错跳步差分格式的图示

如果变量处于适当的半网孔点的中心，那么形如方程(19.1.20)的交错跳步差分会更加清楚明了。记：

$$r_{j+1/2}^n \equiv v\left.\frac{\partial u}{\partial x}\right|_{j+1/2}^n = v\frac{u_{j+1}^n - u_j^n}{\Delta x}$$
$$s_j^{n+1/2} \equiv \left.\frac{\partial u}{\partial t}\right|_j^{n+1/2} = \frac{u_j^{n+1} - u_j^n}{\Delta t} \tag{19.1.34}$$

这纯粹是为了记法上的便利。我们可认为这是定义 $r$ 和 $s$ 的网格，它比定义初始变量 $u$ 的网格加细了一倍。方程(19.1.20)的跳步差分为：

$$\frac{r_{j+1/2}^{n+1} - r_{j+1/2}^n}{\Delta t} = \frac{s_{j+1}^{n+1/2} - s_j^{n+1/2}}{\Delta x}$$
$$\frac{s_j^{n+1/2} - s_j^{n-1/2}}{\Delta t} = v\frac{r_{j+1/2}^n - r_{j-1/2}^n}{\Delta x} \tag{19.1.35}$$

若将方程(19.1.22)代入方程(19.1.35)中，就会发现：又是 Courant 条件为稳定性所需，那

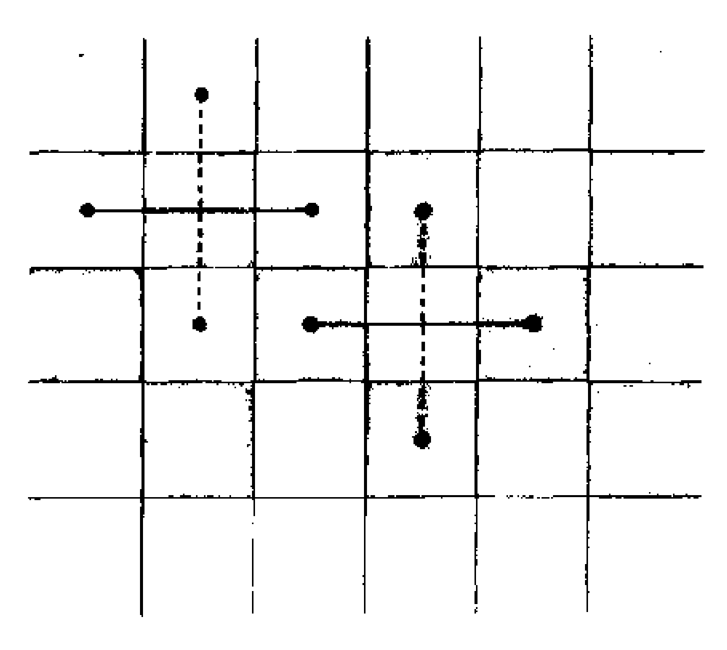
么当该条件被满足后，将不会出现振幅发散。

如果把方程(19.1.34)代入方程(19.1.35)中，便会发现方程(19.1.35)等价于：

$$\frac{u_j^{n+1}-2u_j^n+u_j^{n-1}}{(\Delta t)^2}=v^2\frac{u_{j+1}^n-2u_j^n+u_{j-1}^n}{(\Delta x)^2} \tag{19.1.36}$$

这正是波动方程(19.1.2)的"通常"的二阶差分形式。我们还发现这是一个双层格式，因为为获得 $u^{n+1}$ 需已知 $u^n$ 和 $u^{n-1}$，这一点在方程(19.1.35)中表现为：为改进解，需知 $s^{n-1/2}$ 和 $r^n$。

对于比这简单的典型方程更复杂的方程，特别对于非线性方程，这种方法在梯度变大时，通常变得不稳定。这种不稳定性与奇偶网格点的完全解耦现象有关，如同图19.1.6所示的一个棋盘的黑白方块。通过一个数值粘度项耦合了奇偶网孔，就可以克服这种网孔漂移的不稳性，这粘度项例如在式(19.1.31)右端加入一个小系数(<<1)乘以 $u_{j+1}^n-2u_j^n+u_{j-1}^n$。至于通过增加数值发散使差分格式稳定的方法，参见[4]。



如果网格点被假想为放在一个棋盘的方块中，那么白方格与白方格耦合，黑方格与黑方格耦合；却没有黑白方格之间的耦合。修正的方法是引入一个小的扩散的网孔耦合项。

图19.1.6　在交错跳步格式中网孔漂移不稳定性的起因

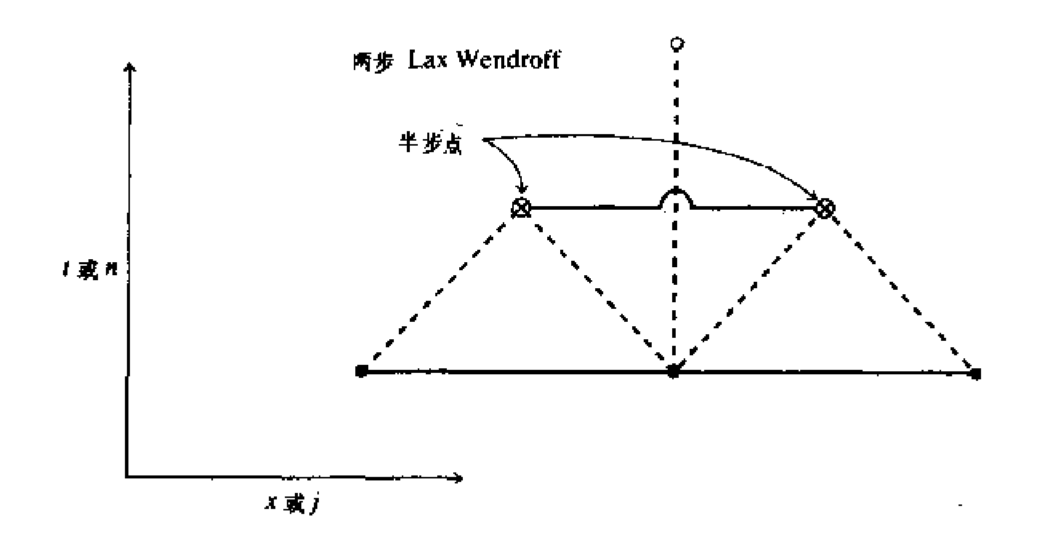
两步 Lax-Wendroff 格式是一种可以避免大的数值发散和网孔飘移的、在时间域上二阶精确的方法。人们在半个时间层 $t_{n+1/2}$ 和半个网孔点 $x_{j+1/2}$ 处定义中间值 $u_{j+1/2}$，通过 Lax 格式计算为：

$$u_{j+1/2}^{n+1/2}=\frac{1}{2}(u_{j+1}^n+u_j^n)-\frac{\Delta t}{2\Delta x}(F_{j+1}^n-F_j^n) \tag{19.1.37}$$

利用这些变量可以计算通量 $F_{j\pm1/2}^{n+1/2}$，然后新值 $u_j^{n+1}$ 可通过适当整理得到的表达式来计算：

$$u_j^{n+1}=u_j^n-\frac{\Delta t}{\Delta x}(F_{j+1/2}^{n+1/2}-F_{j-1/2}^{n+1/2}) \tag{19.1.38}$$

临时值 $u_{j\pm1/2}^{n+1/2}$ 现在就被舍弃了(见图19.1.7)。




两个半层上的点(⊗)用 Lax 方法计算,这两点加上起始点中的一个,运用交错跳步法,可计算新点。半步上的点仅暂时使用,不需要网格点上的存储分配。这种格式在时间和空间上均是二阶精确的。

图19.1.7　两步 Lax—Wendroff 差分格式的图示

让我们考虑一下对于我们的典型对流方程这种方法的稳定性。其中 $F=vu$,把式(19.1.37)代入式(19.1.38)得:

$$u_j^{n+1}=u_j^n-\alpha\Big[\frac{1}{2}(u_{j+1}^n+u_j^n)-\frac{1}{2}\alpha(u_{j+1}^n-u_j^n)-\frac{1}{2}(u_j^n+u_{j-1}^n)+\frac{1}{2}\alpha(u_j^n-u_{j-1}^n)\Big] \tag{19.1.39}$$

其中

$$\alpha\equiv\frac{v\Delta t}{\Delta x} \tag{19.1.40}$$

由此可得:

$$\xi=1-i\alpha\sin k\Delta x-\alpha^2(1-\cos k\Delta x) \tag{19.1.41}$$

故有:

$$|\xi|^2=1-\alpha^2(1-\alpha^2)(1-\cos k\Delta x)^2 \tag{19.1.42}$$

因此稳定性准则 $|\xi|^2\leqslant 1$,即是 $\alpha^2\leqslant 1$ 或如通常的 $v\Delta t\leqslant\Delta x$。顺便说一下,不能认为 Courant 条件是出现偏微分方程中的唯一的稳定性要求。之所以在我们的典型例子中,它一直是稳定性条件,仅仅因为我们的例子在形式上都太简单了。不过,分析的方法是通用的。

除了 $\alpha=1$ 的时候,式(19.1.42)中均有 $|\xi|^2<1$,因此会出现一些振幅衰减。不过由于波长是足以与网格大小 $\Delta x$ 相比拟,因此这处影响相对较小。如果我们对小的 $k\Delta x$ 展开(19.1.42),我们可得:

$$|\xi|^2=1-\alpha^2(1-\alpha^2)\frac{(k\Delta x)^4}{4}+\cdots \tag{19.1.43}$$

对1的偏离仅出现在 $k$ 的 4 次方处。该式应与使用 Lax 方法所得的等式(19.1.16)对比,由该式可得,对小的 $k\Delta x$ 有:

$$|\xi|^2=1-(1-\alpha^2)(k\Delta x)^2+\cdots \tag{19.1.44}$$

总的说来，我们建议对凡可归入流量守恒方程形式的初值问题，或使用交错跳步方法，或使用两步 Lax-Wendorff 方法。对那些对传导误差敏感的问题，迎风差分或它的一种改进可以考虑使用。

### 19.1.5 含有激波的流体动力学

前面我们曾提过，含有激波的流体动力学问题的处理，是一个非常复杂和繁琐的课题。我们在这里，只想给读者介绍一些和该课题有关的基本内容。

激波的处理基本上有三个重要的一般性方法。最古老也是最简单的方法是由冯·诺曼(Von Neumann)和 Richtmyer 提出来的，该方法是给方程加上**模拟粘性**来模拟真实粘性对不连续体的光滑方式。实现该方法的一个较好的出发点是，文献[1]中第12.11节中的差分格式。该格式对几乎所有的一维空间问题都非常适合。

第二种方法是一个高阶差分格式和一个低阶差分格式的结合，前者适用于平稳流，而后者相当耗散并能滤除激波。特别是，各种不同的迎风差分格式的结合使用，除非极陡的梯度出现，否则其权数的选取一般能将低阶格式调整为零，而且这些权数的选取也能施加各种"单调性"限制，从而防止数值解中出现非物理性振动。从文献[2,3,5]来研究这些方法，效果一定很好。

第三种，可能也是最有效的一种方法是 Godunov 法。这里我们抛弃基于泰劳级数的有限差分法中所蕴含的简单的线性化思想，而直接处理方程的非线性。对于由一个不连续体分隔的流体的两个均匀状态的演化问题，即 Riemann 激波问题有解析解。Godunov 的想法是，用大量均匀状态的单体来近似地表示流体，并利用 Riemann 解将其接合起来。Godunov 法现在已有很多的形式，其中最有效的可能是 PPM 法[6]。对所有这些方法的评述。以及对一维方法推广到多维时的讨论，请阅参考文献[7～9]。

参考文献和进一步读物：


Ames, W.F. 1977, *Numerical Methods for Partial Differential Equations*, 2nd ed. (New York: Academic Press), Chapter 4.

Richtmyer, R.D., and Morton, K.W. 1967, *Difference Methods for Initial Value Problems*, 2nd ed. (New York: Wiley-Interscience). [1]

Centrella, J., and Wilson, J.R. 1984, *Astrophysical Journal Supplement*, vol. 54, pp. 229～249, Appendix B. [2]

Hawley, J.F., Smarr, L.L., and Wilson, J.R. 1984, *Astrophysical Journal Supplement*, vol. 55, pp. 211～246, §2c. [3]

Kreiss, H.O. 1978, *Numerical Methods for Solving Time-Dependent Problems for Partial Differential Equations* (Montreal: University of Montreal Press), pp. 66ff. [4]

Harten, A., Lax, P.D., and Van Leer, B. 1983, *SIAM Review*, vol. 25, pp. 36～61. [5]

Woodward, P., and Colella, P. 1984, *Journal of Computational Physics*, vol. 54, pp. 174～201. [6]

Roache, P.J. 1976, *Computational Fluid Dynamics* (Albuquerque: Hermosa). [7]

Woodward, P., and Colella, P. 1984, *Journal of Computational Physics*, vol. 54, pp. 115～173. [8]

Rizzi, A., and Engquist, B. 1987, *Journal of Computational Physics*, vol. 72, pp. 1～69. [9]

## 19.2 扩散初值问题

回顾典型的抛物型方程，一维空间上的扩散方程为：

$$\frac{\partial u}{\partial t}=\frac{\partial}{\partial x}\left(D\frac{\partial u}{\partial x}\right) \tag{19.2.1}$$

其中 $D$ 为扩散系数。事实上，该方程是一个在上节讨论过的通量守恒方程，具有 $x$ 方向的流量：

$$F=-D\frac{\partial u}{\partial x} \tag{19.2.2}$$

我们将假设 $D\geqslant 0$，否则方程(19.2.1)具有物理上的不稳定解：一个小的扰动逐渐变得越来越集中，而不是发散开去。(不要徒劳地去为一个偏微分方程自身就不稳定的问题，去寻找稳定的差分格式。)

尽管式(19.2.1)是一种已经讨论过的形式，但它还是经常被当作一种以其自身方式出现的典型范例。流量(19.2.2)的特殊形式和它的直接推广形式，在实际运用中出现得非常频繁。更进一步地，在许多其它情形下，我们已经看到，数值粘度和人工粘度也可能引入象(19.2.1)右端所示的扩散项。

首先考虑 $D$ 是常数时，方程

$$\frac{\partial u}{\partial t}=D\frac{\partial^2 u}{\partial x^2} \tag{19.2.3}$$

可用一种明显的方式进行差分得

$$\frac{u_j^{n+1}-u_j^n}{\Delta t}=D\left[\frac{u_{j+1}^n-2u_j^n+u_{j-1}^n}{(\Delta x)^2}\right] \tag{19.2.4}$$

除了方程右端进行差分的是一个二阶导数以外，上式即又是FTCS格式了。但这却有了天壤之别。FTCS格式对双曲型方程是不稳定的，而一个快速的计算将表明，方程(19.2.4)中的放大因子为：

$$\xi=1-\frac{4D\Delta t}{(x\Delta)^2}\sin^2\left(\frac{k\Delta x}{2}\right) \tag{9.2.5}$$

由 $|\xi|\leqslant 1$ 的要求可得稳定性准则：

$$\frac{2D\Delta t}{(\Delta x)^2}\leq 1 \tag{19.2.6}$$

限制条件(19.2.6)的物理解释是：允许的最大时间步长是穿过宽度为 $\Delta x$ 的方格的扩散时间，直至一个数值因子。

更一般地，穿过一个大小为 $\lambda$ 的空间区域的扩散时间 $\tau$ 的量级为：

$$\tau\sim\frac{\lambda^2}{D} \tag{19.2.7}$$

通常，我们对精确地模拟 $\lambda>>\Delta x$ 的空间区域中的特征的演变很感兴趣。如果我限制时间步长满足式(19.2.6)，在获得我们感兴趣的区域中的有关信息之前，我们必须通过量级为 $\lambda^2/\Delta x^2$ 的步数来发展。这个步数通常是多得不能运行的。因此，我们必须寻找一种稳定的方法，其中时间步长可与式(19.2.7)的时间量级相比拟；或者，为精确起见，一定程度地小于式(19.2.7)的时间量级。

这一目标提出了一个直接的"哲学"问题。显然，我们建议采用的大的时间步长，对于我们不感兴趣的小区域是极不精确的，而我们希望那些区域中的性能稳定、"无害"，并且可能的话不应过分地在物理上不合理。我们想把这种无害性能加入我们的差分格式中，它又将是怎样的呢？

有两种不同的答案，每个答案有其自身的优点和缺点。第一种方法是寻找一种差分格式，使小区域中的特征达到其**平衡**形式，例如，满足左端置为0的方程(19.2.3)。这种方法一般具有最好的物理意义，但正如我们将看到的，它只能对应一种在我们感兴趣的区域中，关于时间域上是**一阶**精确的差分格式("全隐式")。第二种方法是，让小范围中的特征保持其初振幅值，以使得我们感兴趣的较大区域中的特征变化，与小区域中的"冻结"(虽然有起伏)背景相迭加地进行。这种方法提供了一种在时间域上**二阶**精确的差分格式("Crank-Nicholson"格式)，不过在一个变化过程的计算快结束时，人们可能希望转换为上一种方法的一些时间层，以使小范围中的状态达到平衡。现在，让我们看一下这些独特的差分方法从何得来：

考虑方程(19.2.3)的如下差分式：

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = D\left[\frac{u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}}{(\Delta x)^2}\right] \tag{19.2.8}$$

除了右端的空间导数是估算的第 $n+1$ 时间层外，该式完全就同 FTCS 差分格式一样。这种特征的格式与 FTCS(称为**全显式**)相对照，被称为**全隐式**或**时间后移式**。为了求解方程(19.2.8)，人们不得不在每个时间层解一个联立的线性方程组以求得 $u_j^{n+1}$。幸运的是，该系统是三对角型的，所以，这只是一个简单的问题：只需适当合并方程(19.2.8)中的项：

$$-\alpha u_{j-1}^{n+1} + (1+2\alpha)u_j^{n+1} - \alpha u_{j+1}^{n+1} = u_j^n, \quad j = 1,2,\ldots,J-1 \tag{19.2.9}$$

其中

$$\alpha \equiv \frac{D\Delta t}{(\Delta x)^2} \tag{19.2.10}$$

加上 $j=0$ 和 $j=J$ 处的狄利克莱边界条件或诺伊曼边界条件，方程(19.2.9)是一个三角型系统，它在每个时间层都很容易用第2.6中的方法求解。

对于非常大的时间步长，式(19.2.8)的情形如何呢？当限极 $\alpha \to \infty(\Delta t \to \infty)$，在式(19.2.9)中答案可以看得很清楚：式(19.2.9)两边除以 $\alpha$，我们发现该差分方程就是平衡方程

$$\frac{\partial u}{\partial x^2} = 0 \tag{19.2.11}$$

的有限差分形式。

该差分方法稳定性如何？方程(19.2.8)的放大因子是：

$$\xi = \frac{1}{1 + 4\alpha \sin^2\left(\frac{k\Delta x}{2}\right)} \tag{19.2.12}$$

显然对任意步长 $\Delta t$ 有 $|\xi| < 1$，这种格式是无条件地稳定的。从初始条件发展变化的小区域的细节，对于大的 $\Delta t$，显然是不精确的。但正如前面已经证明的，正确的平衡解已经获得，这也正是隐式方法的特征所在。

另一方面，通过把隐式方法的稳定性和在时间、空间上均二阶精确的方法的精确性综合起来，让我们看看如何满足上述富有哲理性答案的第二个方面。直接把显式和隐式 FTCS 格

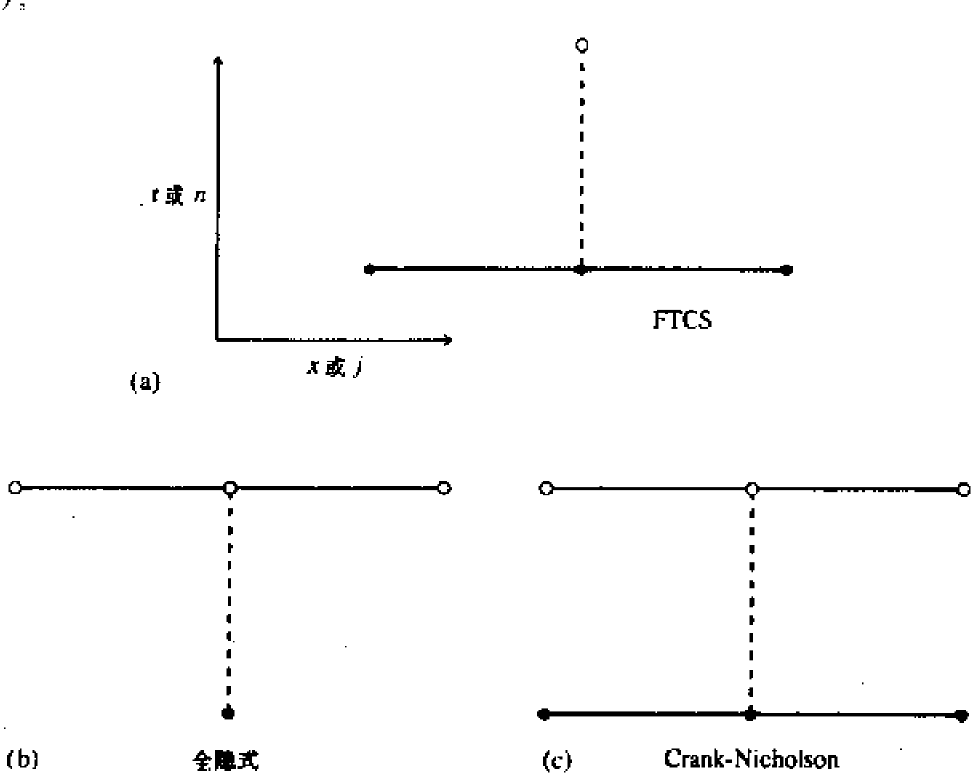
式平均得：

$$\frac{u_j^{n+1}-u_j^n}{\Delta t}=\frac{D}{2}\left[\frac{(u_{j+1}^{n+1}-2u_j^{n+1}+u_{j-1}^{n+1})+(u_{j+1}^n-2u_j^n+u_{j-1}^n)}{(\Delta x)^2}\right] \tag{19.2.13}$$

其中，左边和右边都以时间层 $n+\frac{1}{2}$ 为中心，故这种方法正如前面所称，是时间域上二阶精确的。其放大因子：

$$\xi=\frac{1-2\alpha\sin^2\left(\frac{k\Delta x}{2}\right)}{1+2\alpha\sin^2\left(\frac{k\Delta x}{2}\right)} \tag{19.2.14}$$

故这种方法对任意大小的 Δt 均是稳定的。这种格式被称为 **Crank—Nicholson** 格式，也正是我们对任意的简单扩散问题要推荐的格式(可能在最后几层要补充一些全隐式步子)(见图 19.2.1)。




(a)是时间向前空间中心差分格式，它是一阶精确的，但只对足够小的时间步长稳定；(b)是全隐式格式，它对任意大的时间步长均稳定，但仍然只是一阶精确的；(c)是 Crank—Nicholson 格式，该格式是二阶精确的，且对大的时间步长也通常是稳定的。

图19.2.1　扩散问题的三种差分式(图示方法同于图19.1.2)

现在转向简单扩散方程(19.2.3)的一些一般形式。首先，假设扩散系数 $D$ 不是常数，设 $D=D(x)$，我们可以采纳上述两种方法中的任一种。首先，作一个变量的解析变换：

$$y=\int\frac{dx}{D(x)} \tag{19.2.15}$$

那么

$$\frac{\partial u}{\partial t}=\frac{\partial}{\partial x}D(x)\frac{\partial u}{\partial x} \tag{19.2.16}$$

变成了

$$\frac{\partial u}{\partial t}=\frac{1}{D(y)}\frac{\partial^2 u}{\partial y^2} \tag{19.2.17}$$

对适当的 $y_j$，可以估算 $D_j$由此可得，显式格式中的稳定性准则(19.2.6)变成了：

$$\Delta t\leq\min_j\left[\frac{(\Delta y)^2}{2D_j^{-1}}\right] \tag{19.2.18}$$

注意，沿 $y$ 是常数步长，并不意味着沿 $x$ 也是常数步长。

一种并不要求 $D$ 易解析处理形式的替代方法是，对扩散方程(19.2.16)很简单地差分，在该方法中，对每个变量适当地集中，从而 FTCS 格式变成了：

$$\frac{u_j^{n+1}-u_j^n}{\Delta t}=\frac{D_{j+1/2}(u_{j+1}^n-u_j^n)-D_{j-1/2}(u_j^n-u_{j-1}^n)}{(\Delta x)^2} \tag{19.2.19}$$

其中

$$D_{j+1/2}\equiv D(x_{j+1/2}) \tag{19.2.20}$$

相应的稳定性准则是：

$$\Delta t\leq\min_j\left[\frac{(\Delta x)^2}{2D_{j+1/2}}\right] \tag{19.2.21}$$

故 Crank-Nicholson 方法可以被很容易地推广。

第二种可以考虑的复杂情况是非线性扩散问题，例如方程 $D=D(u)$。显式格式可用简单的方法推广，如在方程(19.2.19)中，记

$$D_{j+1/2}=\frac{1}{2}\left[D(u_{j+1}^n)+D(u_j^n)\right] \tag{19.2.22}$$

隐式格式就不这么简单了。式(19.2.22)中用 $n\to n+1$ 的替换，在每个时间层都留给我们一组难以处理的耦合的非线性方程，通常有一种更简单的方法：设 $D(u)$的形式允许我们对 $z(u)$解析地积分：

$$dz=D(u)du \tag{19.2.23}$$

式(19.2.1)的右端将变为$\partial^2 z/\partial x^2$，可对之隐式差分为：

$$\frac{z_{j+1}^{n+1}-2z_j^{n+1}+z_{j-1}^{n+1}}{(\Delta x)^2} \tag{19.2.24}$$

现在对方程(19.2.24)右端的每项线性化，例如：

$$\begin{aligned}z_j^{n+1}&\equiv z(u_j^{n+1})=z(u_j^n)+(u_j^{n+1}-u_j^n)\left.\frac{\partial z}{\partial u}\right|_{j,n}\\&=z(u_j^n)+(u_j^{n+1}-u_j^n)D(u_j^n)\end{aligned} \tag{19.2.25}$$

这使问题简化成三对角形式，而且在实际问题，通常也保持了隐式差分的稳定性优势。

### 19.2.1 薛定谔方程

有时候，尚待解决的物理问题给我们还未考虑的差分格式加了限制。比如，在量力子学中，依赖于时间的薛定谔(Schrödinger)方程，这是一个关于复数量 $\psi$ 演变的典型双曲型方程。由于每个波群以一维位势 $V(x)$发散，方程具有形式：

$$i\frac{\partial \psi}{\partial t} = -\frac{\partial^2 \psi}{\partial x^2} + V(x)\psi \tag{19.2.26}$$

(在此我们选择了单位使普郎克(Plank)常数 $h=1$,质点质量 $m=1/2$)。已知初始波群 $\psi(x,t=0)$,并已知边界条件为 $x\to\pm\infty$时,$\psi\to 0$。假设我们满足于时间域上一阶精确度,但希望使用隐式格式以求稳定,则将式(19.2.8)稍作推广得:

$$i\left[\frac{\psi_j^{n+1}-\psi_j^n}{\Delta t}\right] = -\left[\frac{\psi_{j+1}^{n+1}-2\psi_j^{n+1}+\psi_{j-1}^{n+1}}{(\Delta x)^2}\right] + V_j\psi_j^{n+1} \tag{19.2.27}$$

其中

$$\xi = \frac{1}{1+i\left[\frac{4\Delta t}{(\Delta x)^2}\sin^2\left(\frac{k\Delta x}{2}\right)+V_j\Delta t\right]} \tag{19.2.28}$$

这是无条件稳定的,但遗憾的却不是幺正的。基本的物理问题要求,在某处发现微粒的总概率应保持为单位 1。这一点可以通过 $\psi$ 的模平方范数保持为 1 来正式地表示:

$$\int_{-\infty}^{+\infty}|\psi|^2 dx = 1 \tag{19.2.29}$$

将初始波函数 $\psi(x,0)$归一化以满足(19.2.29)。于是,薛定谔方程(19.2.26)便保证了在以后任意时刻,这一条件均满足。

让我们把方程(19.2.26)写成如下形式:

$$i\frac{\partial \psi}{\partial t} = H\psi \tag{19.2.30}$$

其中,算子 $H$ 为:

$$H = -\frac{\partial^2}{\partial x^2} + V(x) \tag{19.2.31}$$

方程(19.2.30)的形式解为:

$$\psi(x,t) = e^{-iHt}\psi(x,0) \tag{19.2.32}$$

其中算子的指数形式由其幂级数展开来定义。

用显式 FTCS 格式近似表示式(19.2.32)为:

$$\psi_j^{n+1} = (1-iH\Delta t)\psi_j^n \tag{19.2.33}$$

其中 $H$ 用沿 $x$ 的中心有限差分的逼近来表示。相比之下,稳定的隐式格式(19.2.27)为:

$$\psi_j^{n+1} = (1-iH\Delta t)^{-1}\psi_j^n \tag{19.2.34}$$

通过展开方程(19.2.32)可以发现,这两种格式均在时间域上是一阶精确的,但式(19.2.33)中的算子和式(19.2.34)中的算子都不是幺正的。

对薛定谔方程差分的正确方法是,采用 Cayley 格式给出 $e^{-iHt}$的有限差分表示,这种格式是二阶精确的,并且是幺正的:

$$e^{-iHt} \simeq \frac{1-\frac{1}{2}iH\Delta t}{1+\frac{1}{2}iH\Delta t} \tag{19.2.35}$$

换言之,有:

$$\left(1+\frac{1}{2}iH\Delta t\right)\psi_j^{n+1} = \left(1-\frac{1}{2}iH\Delta t\right)\psi_j^n \tag{19.2.36}$$

此时一旦用沿 $x$ 的有限差分的近似来代替 $H$，我们便到一个待解的复三对角形系统。这种求解薛定谔方程的方法是稳定的、幺正的，并且在时间和空间上均是二阶精确的。事实上，这种方法恰好就是 Crank-Nicholson 方法。

参考文献和进一步读物：


Goldberg, A., Schey, H.M., and Schwartz, J.L. 1967, *American Journal of Physics*, vol 35, pp. 177～186. [1]

Galbraith, I., Ching, Y.S., and Abraham, E. 1984, *American Journal of Physics*, vol. 52, pp. 60～68. [2]


## 19.3 多维的初值问题

在第19.1节和第19.2节中介绍的针对 1+1 维（一维空间和一维时间）问题的方法，可以很容易地推广到 $N+1$ 维空间，但是，求解相应方程所需的计算能力却是巨大的。如果用 100 个空间网格点解决了一个一维问题，那么，解决具有 100×100 个网格点的二维问题至少需要多于 100 倍的计算量。因此，对多维问题，通常不得不满足于非常勉强的空间解。

让我们详细介绍一点关于求解和检验多维偏微分方程的解的经验：通常读者应在非常小的网格上，如 8×8，运行和检验自己的程序，即便相应结果的精确度差得毫无用处。只有程序被全面调试并证明了稳定性，才可以增大网格点的大小到一个适当的程度，并开始观察其结果。我们实际上听过一些人说：我的程序对于粗糙的网格可能是不稳定的，但我确信，这种不稳定性在较大网格中将会消失。这种观点非常荒谬，它体现了对精确度和稳定性概念的完全混淆。事实上，倒是新的不稳定性有时会在较大网格中出现，而旧的不稳定性却绝对不会（根据我们的经验）消失。

由于不得不采用中等大小的网格，有些人建议运用更高阶的方法，以力图改进精确度，这是非常危险的。除非欲求的解已知是平滑的，并且所要用的更高阶方法已知是非常稳定的，否则绝不能建议采用任何时间域中高于二阶的方法（对一系列的一阶方程组）。对于空间差分，我们建议采用原偏微分方程自身的阶数，可能也允许对空间上的一阶偏微分方程采用二阶空间差分。当提高一种差分方法的阶数，使之高于原偏微分方程的阶数时，将会对差分方程引入伪解。如果伪解恰巧都是指数衰减的，那不会有什么问题；反之，将面临一片混乱。

### 19.3.1 通量守恒方程的 Lax 方法

作为一个例子，让我们看看如何把 Lax 方法(19.1.15)推广到守恒方程的二维情形：

$$\frac{\partial u}{\partial t}=-\nabla\cdot\mathbf{F}=-\left(\frac{\partial F_x}{\partial x}+\frac{\partial F_y}{\partial y}\right)\tag{19.3.1}$$

采用空间网格：

$$\begin{aligned}x_j&=x_0+j\Delta\\ y_l&=y_0+l\Delta\end{aligned}\tag{19.3.2}$$

为简单起见，我们选择了 $\Delta y=\Delta x=\Delta$；故 Lax 格式为：

$$u_{j,l}^{n+1}=\frac{1}{4}(u_{j+1,l}^n+u_{j-1,l}^n+u_{j,l+1}^n+u_{j,l-1}^n)$$

$$-\frac{\Delta t}{2\Delta}(F^n_{j+1,l} - F^n_{j-1,l} + F^n_{j,l+1} - F^n_{j,l-1}) \tag{19.3.3}$$

注意，作为缩写符号，$F_{j+1}$和$F_{j-1}$对应$F_x$，而$F_{l+1}$和$F_{l-1}$对应的是$F_y$。

让我们对这个典型对流方程(类似式(19.1.6))进行稳定性分析，设

$$F_x = v_x u, \qquad F_y = v_y u \tag{19.3.4}$$

这使每个特征模虽然在时间上仍简单地依赖于$\xi$的方幂，但在空间中却是两维数了：

$$u^n_{j,l} = \xi^n e^{ik_x j\Delta} e^{ik_y l\Delta} \tag{19.3.5}$$

代入方程(19.3.3)，我们得到：

$$\xi = \frac{1}{2}(\cos k_x\Delta + \cos k_y\Delta) - i\,\alpha_x \sin k_x\Delta - i\,\alpha_y \sin k_y\Delta \tag{19.3.6}$$

其中

$$\alpha_x = \frac{v_x \Delta t}{\Delta}, \qquad \alpha_y = \frac{v_y \Delta t}{\Delta} \tag{19.3.7}$$

$|\xi|^2$的表达式可以被整理成如下形式：

$$|\xi|^2 = 1 - (\sin^2 k_x\Delta + \sin^2 k_y\Delta)\left[\frac{1}{2} - (\alpha_x^2 + \alpha_y^2)\right] - \frac{1}{4}(\cos k_x\Delta - \cos k_y\Delta)^2 - (\alpha_y \sin k_x\Delta - \alpha_x \sin k_y\Delta)^2 \tag{19.3.8}$$

最后两项是负的，所以稳定性要求$|\xi|^2 \leqslant 1$成为

$$\frac{1}{2} - (\alpha_x^2 + \alpha_y^2) \geqslant 0 \tag{19.3.9}$$

或者

$$\Delta t \leq \frac{\Delta}{\sqrt{2}(v_x^2 + v_y^2)^{1/2}} \tag{19.3.10}$$

下面是关于$N$维 Courant 条件的一般形式的一个例子：设$|v|$是该问题中最大的传播速度，则有

$$\Delta t \leq \frac{\Delta}{\sqrt{N}|v|} \tag{19.3.11}$$

为 Courant 条件。

### 19.3.2 多维的扩散问题

让我们考虑一下二维扩散方程

$$\frac{\partial u}{\partial t} = D\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \tag{19.3.12}$$

一种显式方法，如 FTCS，可以从一维情形很容易地进行推广。然而，我们已经知道，扩散问题通常最好是隐式地处理。假设我们力图使用二维的 Crank－Nicholson 格式，将有：

$$u^{n+1}_{j,l} = u^n_{j,l} + \frac{1}{2}\alpha\left(\delta_x^2 u^{n+1}_{j,l} + \delta_x^2 u^n_{j,l} + \delta_y^2 u^{n+1}_{j,l} + \delta_y^2 u^n_{j,l}\right) \tag{19.3.13}$$

其中

$$\alpha \equiv \frac{D\Delta t}{\Delta^2} \qquad \Delta = \Delta x = \Delta y \tag{19.3.14}$$

$$\delta_x^2 u_{j,l}^n = u_{j+1,l}^n - 2u_{j,l}^n + u_{j-1,l}^n \tag{19.3.15}$$

对 $\delta_y^2 u_{j,l}^n$ 有类似的式子。这确实是一种可行的格式，而问题出现在求解耦合的线性方程组的时候。尽管矩阵仍是非常稀疏的，但一维空间中的三对角型系统却不复存在了。一种可能性是使用适当的稀疏矩阵方法(见第2.7节和第19.0节)。

另一种可能的解法，也是我们一般更喜欢使用的，是对一般 Crank-Nicholson 算法的稍微变形。它仍是时间和空间上二阶精确的，并且是无条件稳定的，但其方程比起式(19.3.13)要容易解些。这种格式被称作**交替方向隐式法**(ADI)，其中包含了**算子分裂**或**时间分裂**这种非常有用的思想，我们将在后面讨论更多细节。在此我们须知道，这种思想是旨在把每个时间层以 $\Delta t/2$ 的大小分裂成两层，在每个子层，沿不同方向进行隐式处理。

$$\begin{aligned} u_{j,l}^{n+1/2} &= u_{j,l}^n - \frac{1}{2}\alpha(\delta_x^2 u_{j,l}^{n+1/2} - \delta_y^2 u_{j,l}^n) \\ u_{j,l}^{n+1} &= u_{j,l}^{n+1/2} - \frac{1}{2}\alpha(\delta_x^2 u_{j,l}^{n+1/2} - \delta_y^2 u_{j,l}^{n+1}) \end{aligned} \tag{19.3.16}$$

这种方法的优点是，在每个子层仅需要求解一个简单的三对角型系统。

### 19.3.1 一般算子分裂法

算子分裂，也称为**时间分裂**或**分数步法**，其中心思想是，假定有一个初值方程形为，

$$\frac{\partial u}{\partial t} = \mathscr{L}u \tag{19.3.17}$$

其中 $\mathscr{L}$ 是某个算子。虽然 $\mathscr{L}$ 不一定必须是线性的，但假定它至少可以写成 $m$ 个线性部分的和，叠加作用到 $u$ 上，

$$\mathscr{L}u = \mathscr{L}_1 u + \mathscr{L}_2 u + \cdots + \mathscr{L}_m u \tag{19.3.18}$$

最后，假定对于每个线性部分，已经知道从时间步 $n$ 到时间步 $n+1$ 能修改变量 $u$ 的一种差分格式，而且只要该线性部分是右端唯一的一项，这种差分格式就可使用。我们将这些修改形式写为

$$\begin{aligned} u^{n+1} &= \mathscr{U}_1(u^n, \Delta t) \\ u^{n+1} &= \mathscr{U}_2(u^n, \Delta t) \\ &\cdots \\ u^{n+1} &= \mathscr{U}_m(u^n, \Delta t) \end{aligned} \tag{19.3.19}$$

现在，算子分裂的一种形式，通过从 $n$ 到 $n+1$ 的下述修改序列得到：

$$\begin{aligned} u^{n+(1/m)} &= \mathscr{U}_1(u^n, \Delta t) \\ u^{n+(2/m)} &= \mathscr{U}_2(u^{n+(1/m)}, \Delta t) \\ &\cdots \\ u^{n+1} &= \mathscr{U}_m(u^{n+(m-1)/m}, \Delta t) \end{aligned} \tag{19.3.20}$$

例如，一个复合对流-扩散方程

$$\frac{\partial u}{\partial t} = -v\frac{\partial u}{\partial x} + D\frac{\partial^2 u}{\partial x^2} \tag{19.3.21}$$

对该方程中的对流项采用一种显式格式，而对扩散项采用 Crank-Nicholson 或其它隐式格式可能会很有利。

交替方向隐式(ADI)法,见方程(19.3.16),是一个稍微不同的算子分裂的示例。让我们重新解释一下方程(19.3.19),使其具有一个不同的含义:令 $\mathcal{U}_1$,表示一种修改方法,这种方法包括整个算子 $\mathcal{L}$ 的所有代数部分,而它只期望对 $\mathcal{L}_1$ 部分稳定;其余 $\mathcal{U}_2 \cdots \mathcal{U}_m$ 与 $\mathcal{U}_1$ 类似。那么,从 $u^n$ 到 $u^{n+1}$ 得出的一种新方法是

$$
\begin{aligned}
u^{n+1/m} &= \mathcal{U}_1(u^n, \Delta t/m) \\
u^{n+2/m} &= \mathcal{U}_2(u^{n+1/m}, \Delta t/m) \\
&\cdots \\
u^{n+1} &= \mathcal{U}_m(u^{n+(m-1)/m}, \Delta t/m)
\end{aligned}
\tag{19.3.22}
$$

现在式(19.3.22)中,每个分数步的时间步长只是整个时间步长的 $1/m$,因为每个偏微分运算作用在原始算子的所有项上。

对于算子 $\mathcal{L}$,方程(19.3.22)作为一个差分格式虽然不总是,但也通常都是稳定的。事实上根据经验,通常只要使空间导数最高的算子项具有稳定的 $\mathcal{U}_i$ —— 其它 $\mathcal{U}_i$ 可以不稳定就能使整个格式稳定。

正因为这一点,我们的讨论将从初值问题转向边界值问题。这些内容将占据本章的剩余部分。

## 19.4 边界值问题的傅里叶方法和循环约简法

正如在第19.0节所讨论的,大多数边界值问题(如椭圆型方程)可约简为求解如下形式的、大的稀疏线性系统

$$
\mathbf{A} \cdot \mathbf{u} = \mathbf{b} \tag{19.4.1}
$$

或是对线性的边界值方程一次性求解,或是对非线性的边界值方程迭代求解。

当稀疏矩阵具有某种常见的确定形式时,用两种重要的技巧可得方程(19.4.1)的"快速"解法。**傅里叶变换法**直接适用于方程具有空间域上的常数系数时;而**循环约简法**则在一定程度上更具普遍意义,它的适用性与方程是否可分(在"变量分裂"的意义上)这个问题有关。两种方法都要求边界与坐标轴重合。最后,对一些问题,有一种解法是对这两种方法的极有效的组合,叫作 **FACR** 法(**傅里叶分析和循环约简**)。下面我们依次讨论每种方法,并使用带有限差分表达式(19.0.6)的方程(19.0.3)作为典型范例。一般地说,本节的方法比将在第19.5节讨论的简单松驰法速度快;但却不一定比在第19.6节介绍的较为复杂的多网格松驰法速度快。

### 19.4.1 傅里叶变换法

关于 $x$ 和 $y$ 的离散傅里叶逆变换为:

$$
u_{jl} = \frac{1}{JL} \sum_{m=0}^{J-1} \sum_{n=0}^{L-1} \hat{u}_{mn} \, e^{-2\pi ijm/J} \, e^{-2\pi iln/L} \tag{19.4.2}
$$

该式可用 FFT 在每维空间中进行独立地计算,或者运用第12.4节的程序 **fourn** 或第12.5节的程序 **rlft3** 直接计算。同样地有:

$$
\rho_{jl} = \frac{1}{JL} \sum_{m=0}^{J-1} \sum_{n=0}^{L-1} \hat{\rho}_{mn} \, e^{-2\pi ijm/J} \, e^{-2\pi iln/L} \tag{19.4.3}
$$

如果把表达式(19.4.2)和(19.4.3)代入我们的典型问题(19.0.6)中,我们将有:

$$\hat{u}_{mn}\left(e^{2\pi im/J}+e^{-2\pi im/J}+e^{2\pi in/L}+e^{-2\pi in/L}-4\right)=\hat{\rho}_{mn}\Delta^2 \tag{19.4.4}$$

或者

$$\hat{u}_{mn}=\frac{\hat{\rho}_{mn}\Delta^2}{2\left(\cos\dfrac{2\pi m}{J}+\cos\dfrac{2\pi n}{L}-2\right)} \tag{19.4.5}$$

故运用 FFT 方法求解方程(19.0.6)的步骤为:

·用傅里叶变换计算 $\hat{\rho}_{mn}$:

$$\hat{\rho}_{mn}=\sum_{j=0}^{J-1}\sum_{l=0}^{L-1}\rho_{jl}\,e^{2\pi imj/J}\,e^{2\pi inl/L} \tag{19.4.6}$$

·从方程(19.4.5)中计算 $\hat{u}_{mn}$。

·通过傅里叶逆变换(19.4.2)计算 $u_{jl}$。

上述过程对周期性边界问题适用,换言之,解满足:

$$u_{jl}=u_{j+J,l}=u_{j,l+L} \tag{19.4.7}$$

下面考虑在四边形边界上,狄利克莱边界条件 $u=0$的情形。此时,我们需要用一个正弦波的展开式代替展开式(19.4.2):

$$u_{jl}=\frac{2}{J}\frac{2}{L}\sum_{m=1}^{J-1}\sum_{n=1}^{L-1}\hat{u}_{mn}\sin\frac{\pi jm}{J}\sin\frac{\pi ln}{L} \tag{19.4.8}$$

它满足在 $j=0$, $J$ 处和 $l=0$, $L$ 处,$u=0$ 的边界条件。如果我们把这个展开式和关于 $\rho_{jl}$的类似的展开式代入方程(19.0.6)中,我们将发现相应的求解过程,与原周期性边界问题的求解过程是平行的:

· 通过正弦变换计算 $\hat{\rho}_{mn}$。

$$\hat{\rho}_{mn}=\sum_{j=1}^{J-1}\sum_{l=1}^{L-1}\rho_{jl}\sin\frac{\pi jm}{J}\sin\frac{\pi ln}{L} \tag{19.4.9}$$

(快速正弦变换算法在第12.3中给出。)

· 利用类似于(19.4.5)的表达式计算 $\hat{u}_{mn}$:

$$\hat{u}_{mn}=\frac{\Delta^2\hat{\rho}_{mn}}{2\left(\cos\dfrac{\pi m}{J}+\cos\dfrac{\pi n}{L}-2\right)} \tag{19.4.10}$$

· 运用逆正弦变换(19.4.8)计算 $u_{jl}$。

如果我们的边界条件非齐次,例如,所有边界处满足 $u=0$,但边界 $x=J\Delta$ 处,$u=f(y)$;此时,我们可在上述解中加入一个解 $u^H$,它满足所要求的边界条件中的齐次方程

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}=0 \tag{19.4.11}$$

在连续的情形下,该解将具有如下形式的表达式:

$$u^H=\sum_n A_n\sinh\frac{n\pi x}{J\Delta}\sin\frac{n\pi y}{L\Delta} \tag{19.4.12}$$

其中 $A_n$ 可以通过 $x=J\Delta$ 处 $u=f(y)$来确定,在离散情形下,我们有:

$$u_{jl}^H=\frac{2}{L}\sum_{n=1}^{L-1}A_n\sinh\frac{\pi nj}{J}\sin\frac{\pi nl}{L} \tag{19.4.13}$$

如果 $f(y=l\Delta)=f_l$，则从逆公式可得 $A_n$

$$A_n = \frac{1}{\sinh\pi n}\sum_{l=1}^{L-1} f_l \sin\frac{\pi n l}{L} \qquad (19.4.14)$$

问题的全解为：

$$u = u_{jl} + u_{jl}^{H} \qquad (19.4.15)$$

通过加入形如(19.2.12)的适当项，我们可以处理任何边界面上的非齐次项。

一种处理非齐次项的更简单的方法是，注意到凡边界项出现在式(19.0.6)的左端时，由于它们是已知的，故总可以被移到方程的右端，所以有效的源项是 $\rho_{jl}$ 加上一个边界项的贡献。

为了形式上实现该想法，将解写为

$$u = u^{I} + u^{B} \qquad (19.4.16)$$

其中在边界上 $u^{I}=0$，而 $u^{B}$ 是除边界外其余处均取零。在边界上，$u^{B}$ 取给定的边界值。在上述示例中，$u^{B}$ 的非零值将只有

$$u^{B}_{J,l} = f_l \qquad (19.4.17)$$

典型方程(19.0.3)变为

$$\nabla^2 u^{I} = -\nabla^2 u^{B} + \rho \qquad (19.4.18)$$

或者，以有限差分的形式表示为

$$\begin{aligned} u^{I}_{j+1,l} + u^{I}_{j-1,l} + u^{I}_{j,l+1} + u^{I}_{j,l-1} - 4u^{I}_{j,l} = \\ -(u^{B}_{j+1,l} + u^{B}_{j-1,l} + u^{B}_{j,l+1} + u^{B}_{j,l-1} - 4u^{B}_{j,l}) + \nabla^2\rho_{j,l} \end{aligned} \qquad (19.4.19)$$

方程(19.4.19)中，由于所有 $u^{B}$ 项除了 $j=J-1$ 处之外均为零，故当 $j=J-1$ 时，得

$$u^{I}_{J,l} + u^{I}_{J-2,l} + u^{I}_{J-1,l+1} + u^{I}_{J-1,l-1} - 4u^{I}_{J-1,l} = -f_l + \nabla^2\rho_{J-1,l} \qquad (19.4.20)$$

因此该问题现在与零边界条件等价，但源项的一行需进行下面的替换

$$\nabla^2\rho_{J-1,l} \to \nabla^2\rho_{J-1,l} - f_l \qquad (19.4.21)$$

诺伊曼边界条件 $\nabla u=0$ 的情形，可通过余弦展开式(12.3.17)来处理：

$$u_{jl} = \frac{2}{J}\,\frac{2}{L}\sum_{m=0}^{J}{}'' \sum_{n=0}^{L}{}'' \hat{u}_{mn}\cos\frac{\pi j m}{J}\cos\frac{\pi l n}{L} \qquad (19.4.22)$$

这里双撇号表示 $m=0$ 及 $m=J$ 的项应该乘以 $\frac{1}{2}$，$n=0$ 及 $n=L$ 也类似。仍然可包含非齐次项 $\nabla u=g$，其处理的方法可以通过加上齐次方程的一个合适的解，或更简单地将边界项移到右端来实现。例如，条件

$$\frac{\partial u}{\partial x} = g(y) \qquad 在\ x=0 \qquad (19.4.23)$$

变为

$$\frac{u_{1,l} - u_{-1,l}}{2\Delta} = g_l \qquad (19.4.24)$$

其中 $g_l \equiv g(y=l\Delta)$。我们再次将解写为(19.4.16)的形式，其中在边界上 $\nabla u^{I}=0$。这次 $\nabla u^{B}$ 在边界上取所规定的值，而 $u^{B}$ 除边界之外均取为零。

因此方程(19.4.24)给出

$$u^{B}_{-1,l} = -2\Delta g_l \qquad (19.4.25)$$

方程(19.4.10)中，所有 $u^B$ 项除了当 $j=0$ 以外均为零，得：

$$u'_{1,l} + u'_{-1,l} + u'_{0,l+1} + u'_{0,l-1} - 4u'_{0,l} = 2\Delta g_l + \Delta^2 \rho_{0,l} \tag{19.4.26}$$

因此 $u'$ 是零梯度问题的解，但源项需进行下面的替换：

$$\Delta^2 \rho_{0,l} \rightarrow \Delta^2 \rho_{0,l} + 2\Delta g_l \tag{19.4.27}$$

有时诺伊曼边界条件是通过利用一个交错网格来处理的，$u$ 定义在区域边界的中间，它们的一阶导数集中在网格点上。如果采用余弦变换的其它形式，例如，方程(12.3.23)，则利用类似于上述的技巧，同样可以解决这类问题。

## 19.4.2 循环约简法

显然，FFT 方法仅当初始偏微分方程具有常系数，并且边界与坐标轴相吻合时适用。一种可用于稍微更为一般的方程的算法，叫做**循环约简法**。

我们以方程

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + b(y)\frac{\partial u}{\partial y} + c(y)u = g(x,y) \tag{19.4.28}$$

为例来阐述循环约简法。在实际运用中，这种方程形式经常由极坐标、柱坐标或球坐标体系下的海姆霍兹 Helmholtz 方程或者泊松方程得到。更一般的可分离方程在[1]中处理。

方程(19.4.28)的有限差分式可以写成一组向量方程的形式：

$$\mathbf{u}_{j-1} + \mathbf{T}\cdot\mathbf{u}_j + \mathbf{u}_{j+1} = \mathbf{g}_j\Delta^2 \tag{19.4.29}$$

其中下标 $j$ 取自沿 $x$ 方向的差分，而 $y$ 方向的差分(由前面约定的下标 $l$ 标记)则保留在向量中。矩阵 $\mathbf{T}$ 形为：

$$\mathbf{T} = \mathbf{B} - 2\mathbf{1}$$

其中 $2\mathbf{1}$ 取自 $x$ 方向的差分，矩阵 $\mathbf{B}$ 取自 $y$ 方向的差分。矩阵 $\mathbf{B}$ 以及由此而得的矩阵 $\mathbf{T}$，均是带有可变系数的三对角型矩阵。

CR 方法通过写出象(19.4.29)的三个相邻方程来导出：

$$\begin{aligned}
\mathbf{u}_{j-2} + \mathbf{T}\cdot\mathbf{u}_{j-1} + \mathbf{u}_j &= \mathbf{g}_{j-1}\Delta^2 \\
\mathbf{u}_{j-1} + \mathbf{T}\cdot\mathbf{u}_j + \mathbf{u}_{j+1} &= \mathbf{g}_j\Delta^2 \\
\mathbf{u}_j + \mathbf{T}\cdot\mathbf{u}_{j+1} + \mathbf{u}_{j+2} &= \mathbf{g}_{j+1}\Delta^2
\end{aligned} \tag{19.4.31}$$

用矩阵 $-\mathbf{T}$ 乘以中间的方程，并把这三个方程加起来，我们得：

$$\mathbf{u}_{j-2} + \mathbf{T}^{(1)}\cdot\mathbf{u}_j + \mathbf{u}_{j+2} = \mathbf{g}_j^{(1)}\Delta^2 \tag{19.4.32}$$

这是一个与式(19.4.29)有相同形式的方程，其中

$$\begin{aligned}
\mathbf{T}^{(1)} &= 2\mathbf{1} - \mathbf{T}^2 \\
\mathbf{g}_j^{(1)} &= \Delta^2(\mathbf{g}_{j-1} - \mathbf{T}\cdot\mathbf{g}_j + \mathbf{g}_{j+1})
\end{aligned} \tag{19.4.33}$$

通过一层 CR 处理，我们将方程的个数减少了一半；由于所得方程与原方程有相同的形式，所以我们可以重复上过程。为简便起见，取网格上数目为 2 的方幂，我们最后结束在一个关于变量中心线的方程上：

$$\mathbf{T}^{(f)}\cdot\mathbf{u}_{J/2} = \Delta^2\mathbf{g}_{J/2}^{(f)} - \mathbf{u}_0 - \mathbf{u}_J \tag{19.4.34}$$

该方程中，由于 $\mathbf{u}_0$、$\mathbf{u}_J$ 是已知的边界值，我们把它们写到右边。方程(19.4.34)可用标准三对角算法求解，以得 $\mathbf{u}_{J/2}$。在 $f-1$ 层的两个方程含 $\mathbf{u}_{J/4}$ 和 $\mathbf{u}_{3J/4}$，含 $\mathbf{u}_{J/4}$ 的方程用到 $\mathbf{u}_0$ 和 $\mathbf{u}_{J/2}$，两

个都是已知的，故可用通常的三对角算法求解。在每一层都有相似的情形，因此我们共需解决 $J-1$ 个三对角型系统。

在实际运用中，方程(19.4.33)应该改写，以避免数值的不稳定性。对于这些问题和其它实际运用中的细节，请参阅文献[2]。

### 19.4.3 FACR 方法

求解形如(19.4.28)的方程，包括式(19.0.3)的常系数问题，最佳方法是傅里叶分析和循环约简的联合使用，即 FACR 方法[3-6]。如果在 CR 方法的第 r 层，我们对形如(19.4.32)的方程沿 $y$ 方向作傅里叶分析，即对被压缩的向量用傅里叶变换，我们将得到一个 $x$ 方向上关于每个 $y$ 傅里叶模的三对角型系统：

$$\hat{u}^k_{j-2^r} + \lambda_k^{(r)}\hat{u}^k_j + \hat{u}^k_{j+2^r} = \Delta^2 g_j^{(r)k} \qquad (19.4.35)$$

其中 $\lambda_k^{(r)}$ 是 $\mathbf{T}^{(r)}$ 中对应于第 $k$ 个傅里叶模的特征值。对于方程(19.0.3)，方程(19.4.5)表明 $\lambda_k^{(r)}$ 将含有象 $\cos(2\pi k/L)-2$ 自乘这样的项。在 $j=2^r, 2\times 2^r, 4\times 2^r, \cdots J-2^r$ 层，求解三对角型系统，得 $\hat{u}^k_j$。再用傅里叶方法综合得到在这些 $x$ 线上的 $y$ 值，然后象在先前的 CR 算法一样，填充中间的 $x$ 线。

这种方法的诀窍在于，适当选取 CR 的层数，使算法的操作步骤可达到最少。可以想见，对于一个典型的 128×128 网格情形，理想的层数 $r=2$；渐近地，有 $r \to \log_2(\log_2 J)$。

对方程(19.0.3)使用上述算法在运行时间上的粗略估计如下：FFT 方法(沿 x 方向和 y 方向)和 CR 方法基本上可以相比拟；FACR 方法在取 r=0（即在一维方向使用 FFT，在另一维方向用通常算法求解三对角线方程组）时，可在速度上约提高两倍；理想的取 r=2 的 FACR 方法则可以在速度上再提高两倍。

参考文献和进一步读物：


Swartzrauber, P. N. 1977, *SIAM Review*, vol. 19, pp. 490~501. [1]

Buzbee, B. L., Golub, G. H., and Nielson, C. W. 1970, *SIAM Journal on Numerical Analysis*, vol. 7, pp. 627-656; see also *op. cit.* vol. 11, pp. 753~763. [2]

Hockney, R. W. 1965, *Journal of the Association for Computing Machinery*, vol. 12, pp. 95~113. [3]

Hockney, R. W. 1970, in *Methods of Computational Physics*, vol. 9 (New York: Acadmic Press), pp. 135~211. [4]

Hockney, R. W., and Eastwood, J. W. 1981, *Computer Simulation Using Particles* (New York: McGraw-Hill), Chapter 6. [5]

Temperton, C. 1980, *Journal of Computational Physics*, vol. 34, pp. 314~329. [6]


## 19.5 边界值问题的松弛法

正如我们在第19.0节提到的，松驰法涉及到分裂由有限差分产生的稀疏矩阵，然后迭代运算直到解被发现。

还有一种考虑松弛法的方式，它在一定程度上更具物理意义。假设我们想求解椭圆方程

$$\mathscr{L}u = \rho \qquad (19.5.1)$$

其中 $\mathscr{L}$ 表示某些椭圆算子，$\rho$ 是源项，改写方程为扩散方程

$$\frac{\partial u}{\partial t} = \mathscr{L}u - \rho \qquad (19.5.2)$$

当 $t\to\infty$ 时，初始分配 $u$ **松驰**到一个平衡解，这个平衡解在任意时刻导数为 0，因此它就是原椭圆问题(19.5.1)的解。这样一来，在第19.2节中所有关于扩散的初值方程的技巧，均可通过松驰法用边界值问题求解。

让我们把这个思想应用在我们的典型问题(19.0.3)上，扩散方程为：

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} - \rho \qquad (19.5.3)$$

如果我们利用 FTCS 格式进行差分(参见方程19.2.4)，我们得到：

$$u_{j,l}^{n+1} = u_{j,l}^{n} + \frac{\Delta t}{\Delta^2}(u_{j+1,l}^{n} + u_{j-1,l}^{n} + u_{j,l+1}^{n} + u_{j,l-1}^{n} - 4u_{j,l}^{n}) - \rho_{j,l}\Delta t \qquad (19.5.4)$$

回忆一下式(19.2.6)，我们知道，FTCS 差分在一维空间中，仅在 $\Delta t/\Delta^2\leqslant 1/2$ 时是稳定的；在两维空间中，条件变为 $\Delta t/\Delta^2\leqslant 1/4$。假设我们试图选择可能的最大时间步长，即设 $\Delta t = \Delta^2/4$，那么方程(19.5.4)变为：

$$u_{j,l}^{n+1} = \frac{1}{4}(u_{j+1,l}^{n} + u_{j-1,l}^{n} + u_{j,l+1}^{n} + u_{j,l-1}^{n}) - \frac{\Delta^2}{4}\rho_{j,l} \qquad (19.5.5)$$

因此算法中包括利用 $u$ 在网格上的四个最近邻点求平值值(加上来自源项的贡献)。这个过程一直迭代到收敛为此。

这种方法实际上是一种经典方法，其起源可以追溯到上个世纪，被称作**雅可比方法**(注意不要与求特征值的雅可比方法混淆)。由于这种方法收敛很慢，所以它并不实用。不过它是理解现代方法的基础，现代方法多与它进行比较。

另一种经典方法是高斯-塞德尔(Seidel)方法，这种方法在多网格法中很重要。该方法中，一旦式(19.5.5)右端 $u$ 的新值可用，我们就利用它，换言之，求平均是"同址"进行的，而不是由较早时间层"拷贝"到较晚时间层。如果我们沿行进行该过程，即固定 $l$ 值而逐步增大 $j$，我们有

$$u_{j,l}^{n+1} = \frac{1}{4}(u_{j+1,l}^{n} + u_{j-1,l}^{n+1} + u_{j,l+1}^{n} + u_{j,l-1}^{n+1}) - \frac{\Delta^2}{4}\rho_{j,l} \qquad (19.5.6)$$

这种方法也收敛得很缓慢，仅有理论上的价值。不过它的一些分析还是很有启发性的。

让我们从矩阵分裂的角度来看看雅可比方法和高斯-塞霍尔方法。我们改变标记，把 $\mathbf{u}$ 记为："$\mathbf{x}$"，以与标准矩阵记法一致。为解

$$\mathbf{A}\cdot\mathbf{x} = \mathbf{b} \qquad (19.5.7)$$

我们考虑把 $\mathbf{A}$ 分裂为：

$$\mathbf{A} = \mathbf{L} + \mathbf{D} - \mathbf{U} \qquad (19.5.8)$$

其中 $\mathbf{D}$ 是 $\mathbf{A}$ 的对角线部分；$\mathbf{L}$ 是 $\mathbf{A}$ 的下三角部分，对角线上元素取 0；$\mathbf{U}$ 是 $\mathbf{A}$ 的上三角部分，对角线上元素取 0。

在雅可比方法中，我们写出迭代的第 $r$ 步为：

$$\mathbf{D}\cdot\mathbf{x}^{(r)} = -(\mathbf{L} + \mathbf{U})\cdot\mathbf{x}^{(r-1)} + \mathbf{b} \qquad (19.5.9)$$

对于我们的典型问题(19.5.5)，$\mathbf{D}$ 是一个简单的单位矩阵。雅可比方法对那些具有"对角优势"的矩阵 $\mathbf{A}$，在数学精确的意义上收敛，对于由有限差分产生的矩阵，这个条件通常是满足的。

雅可比方法的收敛速度是多少呢？具体的讨论超出了我们的范围，不过，在此还是略谈一二：矩阵 $-\mathbf{D}^{-1}\cdot(\mathbf{L}+\mathbf{U})$ 是**迭代矩阵**，撇开加项不管，它把一组 $\mathbf{x}$ 映射到下一组。这个迭代矩阵有特征值，每个特征值都反映了一个因子，在每次迭代中，正是通常过该因子抑制了非期望残差的一个特殊特征模的振幅。显然，为了松驰法能够确实有效，这些因子最好都满足模<1。雅可比方法的收敛速度由特征模的最慢的衰减速度决定，即具有最大模的因子的速率决定，因此，该最大因子的模应界于0、1之间，被称为松驰算子的**谱半径**，记作 $\rho_s$。

因此，使全局误差降低到原先的$10^{-p}$倍，所需迭代次数 $r$ 可估计为：

$$r \approx \frac{p\ln 10}{(-\ln\rho_s)} \tag{19.5.10}$$

一般说来，随着网格尺寸 $J$ 的增大，谱半径 $\rho_s$ 将逐渐逼近数值 1，这就要求有更多的迭代。对于任意给定的方程、网格形状和**边界条件**，谱半径在原则上可以解析地计算。例如，对于具有 $J\times J$ 网格和在四边均满足狄利克边界条件的方程(19.5.5)，对大的 $J$ 值，$\rho_s$ 的渐近公式可证得为：

$$\rho_s \simeq 1-\frac{\pi^2}{2J^2} \tag{19.5.11}$$

为使误差降低到原来的$10^{-p}$倍，所需迭代次数 $r$ 为：

$$r \simeq \frac{2pJ^2\ln 10}{\pi^2} \sim \frac{1}{2}pJ^2 \tag{19.5.12}$$

换言之，迭代次数与网格点数 $J^2$成比例。由于 100×100 的网格和更大的网格问题有相同的结论，非常显然，雅可比方法只有学术价值。

对于方程(19.5.6)，高斯-塞德尔方法相应于矩阵分解有：

$$(\mathbf{L}+\mathbf{D})\cdot\mathbf{x}^{(r)} = -\mathbf{U}\cdot\mathbf{x}^{(r-1)}+\mathbf{b} \tag{19.5.13}$$

$\mathbf{L}$ 在方程左端，这是可根据适当的修正而得出，如按分量写出(19.5.13)式，很容易检验出这点。人们可以证明[1-3]，该方法的谱半径正好是雅可比方法的谱半径的平方，故对我们的典型问题有：

$$\rho_s \simeq 1-\frac{\pi^2}{J^2} \tag{19.5.14}$$

$$r \simeq \frac{pJ^2\ln 10}{\pi^2} \simeq \frac{1}{4}pJ^2 \tag{19.5.15}$$

这个方法比雅可比法在迭代次数上减少了一半，但是，这仍然不能使该方法变得实用。

### 19.5.1 同步超松驰(SOR)法

如果我们在高斯-塞德尔迭代的第 $r$ 步对 $\mathbf{x}^{(r)}$的值作一个**过度修正**，我们将得到一个更好的算法，它一直到1970年都是作为标准算法。解方程(19.5.13)得 $\mathbf{x}^{(r)}$，并在式(19.5.13)的右端加减 $\mathbf{x}^{(r-1)}$，由此高斯-塞德尔方法可写为：

$$\mathbf{x}^{(r)} = \mathbf{x}^{(r-1)} - (\mathbf{L}+\mathbf{D})^{-1}[(\mathbf{L}+\mathbf{D}-\mathbf{U})\mathbf{x}^{(r-1)}-\mathbf{b}] \tag{19.5.16}$$

在方括号中的项正是残差向量 $\xi^{(r-1)}$，于是有：

$$\mathbf{x}^{(r)} = \mathbf{x}^{(r-1)} - (\mathbf{L}+\mathbf{D})^{-1}\cdot\xi^{(r-1)} \tag{19.5.17}$$

现在进行过度修正，定义

$$\mathbf{x}^{(r)} = \mathbf{x}^{(r-1)} - \omega(\mathbf{L}+\mathbf{D})^{-1}\cdot\xi^{(r-1)} \tag{19.5.18}$$

其中 $\omega$ 被称为**超松驰参数**，这种方法则叫做**同步超松驰**(SOR)**法**。

以下原理可予以证明：

- 仅对 $0 < \omega < 2$，该方法收敛；如果 $0 < \omega < 1$，我们称之为**低松驰**。
- 在有限差分产生的矩阵通常都满足的一些数学限制下，只有超松驰法($1 < \omega < 2$)可使收敛快于高斯-塞德尔方法。
- 设 $\rho_{雅可比}$ 是雅可比迭代的谱半径(其平方即是高斯-塞德尔迭代的谱半径)，则对 $\omega$ 的最佳选择为：

$$\omega = \frac{2}{1 + \sqrt{1 - \rho_{雅可比}^2}} \tag{19.5.19}$$

- 对 $\omega$ 的这个最佳选择，SOR 方法的谱半径为

$$\rho_{\mathrm{SOR}} = \left( \frac{\rho_{雅可比}}{1 - \sqrt{1 - \rho_{雅可比}^2}} \right)^2 \tag{19.5.20}$$

作为以上结果的应用，考虑我们谱半径 $\rho_{雅可比}$ 是由等式(19.5.11)给出的典型问题。由等式(19.5.19)和(19.5.20)得：

$$\omega \simeq \frac{2}{1 - \pi/J} \tag{19.5.21}$$

$$\rho_{\mathrm{SOR}} \simeq 1 - \frac{2\pi}{J} \qquad 对大的 J \tag{19.5.22}$$

等式(19.5.10)给出使初始误差减少到原先的 $10^{-p}$ 倍所需的迭代次数：

$$r \simeq \frac{pJ\ln 10}{2\pi} \simeq \frac{1}{3} pJ \tag{19.5.23}$$

与式(19.5.12)或(19.5.15)对比，我们发现，理想的 SOR 方法不同于 $J^2$ 量级的迭代，只需要 $J$ 量级迭代。由于 $J$ 一般为 100 或者更大，这就产生了巨大的差别。等式(19.5.23)说明，3 位数字精确度($p=3$)需要的迭代次数等于沿网格一边的网点数；而对于 6 位数字精确度，我们需要约两倍那么多次的迭代。

对于那些谱半径不是解析可解的情形，如何选择 $\omega$ 呢？这正是 SOR 方法的弱点！SOR 方法的优势，仅在围绕 $\omega$ 的正确值的一个相当狭窄的窗口中获得，并且宁可把 $\omega$ 选得稍偏大，也不要把它选择得偏小，当然最好还是获得 $\omega$ 的正确值。

一种选择的 $\omega$ 的方法是，把问题近似地与一个已知问题对照起来，用平均值代换方程中的系数。不过要注意，已知的问题必须与实际问题有着相同的网格尺寸和边界条件。下面我们给出，我们的典型问题在矩形的 $J \times L$ 网格上的 $\rho_{雅可比}$ 值以供参考，允许存在 $\Delta x \neq \Delta y$ 的可能性：

$$\rho_{雅可比} = \frac{\cos\dfrac{\pi}{J} - \left(\dfrac{\Delta x}{\Delta y}\right)^2 \cos\dfrac{\pi}{L}}{1 + \left(\dfrac{\Delta x}{\Delta y}\right)^2} \tag{19.5.24}$$

该等式对狄利克莱边界条件或诺伊曼边界条件保持不变，对于周期性边界条件，作代换 $\pi \to 2\pi$。

当想求解许多相似的椭圆方程，且每次只有系数有细微的变化时，下面的一种方法特别有用：考虑对第一个方程凭经验确定其最佳 $\omega$ 值，然后将此值用于其它方程。关于使用该方

法和“搜寻”最佳 $\omega$ 值的各种自动程序，在有关文献中有所介绍。

虽然前面介绍的矩阵标记对理论分析很有用，但对于 SOR 算法的实际运用，我们还是需要显式的格式。考虑一个一般的沿 $x$ 和 $y$ 方向的二阶椭圆方程，如同我们对典型方程一样，在一个正方形中进行有限差分，因此对应于矩阵 A 的每一行，有一个方程形为：

$$a_{j,l}u_{j+1,l}+b_{j,l}u_{j-1,l}+c_{j,l}u_{j,l+1}+d_{j,l}u_{j,l-1}+e_{j,l}u_{j,l}=f_{j,l} \qquad (19.5.25)$$

对我们的典型方程，我们有 $a=b=c=d=1, e=-4$，量 $f$ 与源项成比例，迭代过程由对 $u_{j,l}$ 求解式(19.5.25)的过程来确定：

$$u_{j,l}^{*}=\frac{1}{e_{j,l}}(f_{j,l}-a_{j,l}u_{j+1,l}-b_{j,l}u_{j-1,l}-c_{j,l}u_{j,l-1}-d_{j,l}u_{j,l-1}) \qquad (19.5.26)$$

$u_{j,l}^{新}$是一个加权平均值：

$$u_{j,l}^{新}=\omega u_{j,l}^{*}+(1-\omega)u_{j,l}^{旧} \qquad (19.5.27)$$

我们对 $u_{j,l}^{新}$ 计算如下：任一步的残差是

$$\xi_{j,l}=a_{j,l}u_{j+1,l}+b_{j,l}u_{j-1,l}+c_{j,l}u_{j,l+1}+d_{j,l}u_{j,l-1}+e_{j,l}u_{j,l}-f_{j,l} \qquad (19.5.28)$$

由 SOR 算法式(19.5.18)或(19.5.27)得：

$$u_{j,l}^{新}=u_{j,l}^{旧}-\omega\frac{\xi_{j,l}}{e_{j,l}} \qquad (19.5.29)$$

该公式非常容易编程，且残差向量 $\xi_{j,l}$ 的范数可用作终止迭代的衡量标准。

另一个实际运用中的关键点是：注意网格点排列的顺序。一种直接的策略是简单地沿行（或沿列）排列；另一种方法，假设我们把网格分为“奇”网格和“偶”网格，如一个棋盘上的黑白方块，则方程(19.5.26)即表示奇点仅依赖于偶网格处的值，反之亦然。因此，我们可以完成一半扫描，修正奇点值，并根据这些新的奇点值，完成另一半扫描，修正偶点值。从下面有关 SOR 方法的实施来看，我们将采纳奇—偶排序。

最后一个实际运用中的关键点是：SOR 方法的渐近收敛速度直到 J 阶迭代后才能达到，而在收敛获得之前，误差通常以 20 倍的速度增长。对 SOR 方法的一点小小修正可以解决这个问题。正是基于我们观察可知：尽管 $\omega$ 表示最佳的渐近松驰参数，但它不必是一个好的初始选择值。在具有切比雪夫加速度的 SOR 方法中，利用奇—偶排序，按照如下规则，可在每次一半扫描时改变 $\omega$ 值：

$$\begin{aligned}
\omega^{(0)}&=1\\
\omega^{(1/2)}&=1/(1-\rho_{雅可比}^{2}/2)\\
\omega^{(n+1/2)}&=1/(1-\rho_{雅可比}^{2}\omega(n)/4), \qquad n=1/2,1,\ldots,\infty\\
\omega^{(\infty)}&\to\omega_{最佳值}
\end{aligned} \qquad (19.5.30)$$

切比雪夫加速度的巧妙之处在于，在每次迭代中，误差的范数总在减少（这是指 $u_{j,l}$ 的实际误差范数，残差 $\xi_{j,l}$ 的范数不必单调下降）。尽管此法中，渐近收敛速度与通常 SOR 一样，但为了减少所需迭代的总数，没有任何理由不使用切比雪夫加速度。

下面我们给出具有切比雪夫加速度的 SOR 方法的程序：

```
#include <math.h>
#define MAXITS 1000
#define EPS 1.0e-5
```

```
void sor(double **a, double **b, double **c, double **d, double **e,
    double **f, double **u, int jmax, double rjac)
    方程(19.5.25)的带有切比雪夫加速度的同步超松驰法求解。a,b,c,d,e和f作为方程的系数输入，每个维数为
    网格尺寸[1..jmax][1..jmax]。u作为解的初猜测值输入，常常置为0，以最后结果值返回。rjac作为雅可比迭代的
    谱半径或其估计值输入。
{
    void nrerror(char error_text[]);
    int ipass,j,jsw,l,lsw,n;
    double anorm,anormf=0.0,omega=1.0,resid; 对jmax约大于25时，采用双精度较好


    for (j=2;j<jmax;j++)   计算残差的初始范数并当范数减少了EPS倍时终止迭代


        for (l=2;l<jmax;l++)
            anormf += fabs(f[j][l]);            假设u的初始值为零
    for (n=1;n<=MAXITS;n++) {
        anorm=0.0;
        jsw=1;
        for (ipass=1;ipass<=2;ipass++) {       奇-偶排序
            lsw=jsw;
            for (j=2;j<jmax;j++) {
                for (l=lsw+1;l<jmax;l+=2) {
                    resid=a[j][l]*u[j+1][l]
                        +b[j][l]*u[j-1][l]
                        +c[j][l]*u[j][l+1]
                        +d[j][l]*u[j][l-1]
                        +e[j][l]*u[j][l]
                        -f[j][l];
                    anorm += fabs(resid);
                    u[j][l] -= omega*resid/e[j][l];
                }
                lsw=3-lsw;
            }
            jsw=3-jsw;
            omega=(n == 1 && ipass == 1 ? 1.0/(1.0-0.5*rjac*rjac) :
                1.0/(1.0-0.25*rjac*rjac*omega));
        }
        if (anorm < EPS*anormf) return;
    }
    nrerror("MAXITS exceeded");
}
```

SQR 方法的主要优点是易于编程，其主要缺点是它对大问题仍不是非常有效。

### 19.5.2 交替方向隐式法

第19.3中用于扩散方程的 ADI(交替方向隐式法)可以转换成用于椭圆方程的松弛法。在第19.3节中，我们曾讨论过用 ADI 方法来解时间依赖的热流方程

$$\frac{\partial u}{\partial t} = \nabla^2 u - \rho \qquad (19.5.31)$$

通过设 $t\to\infty$，也可得到用来解椭圆方程

$$\nabla^2 u = \rho \qquad (19.5.32)$$

的一种迭代方法。每种情形的算子分裂都具有形式为

$$\mathscr{L} = \mathscr{L}_x - \mathscr{L}_y \qquad (19.5.33)$$

其中 $\mathscr{L}_x$ 代表对 $x$ 进行差分，$\mathscr{L}_y$ 代表对 $y$ 进行差分。

例如，在我们的典型问题(19.0.6)中，取 $\Delta x = \Delta y = \Delta$，得到

$$\mathscr{L}_x u = 2u_{j,l} - u_{j+1,l} - u_{j-1,l}$$
$$\mathscr{L}_y u = 2u_{j,l} - u_{j,l+1} - u_{j,l-1} \tag{19.5.34}$$

更复杂的算子也可进行类似地分裂，但要涉及一些技巧问题。分裂不当的选择可能会导致不收敛算法的产生。通常，我们是根据问题的物理性质来进行算子分裂的。我们知道对于我们的典型问题，把一个初始瞬变值传播出去，于是我们建立起对 $x$ 和 $y$ 的算子分裂，来模拟每维的扩散。

选定算子分裂法后，我们按两个半步对时间依赖方程(19.5.31)进行隐式差分：

$$\frac{u^{n+1/2} - u^n}{\Delta t/2} = -\frac{\mathscr{L}_x u^{n+1/2} + \mathscr{L}_y u^n}{\Delta^2} - \rho$$
$$\frac{u^{n+1} - u^{n+1/2}}{\Delta t/2} = -\frac{\mathscr{L}_x u^{n+1/2} + \mathscr{L}_y u^{n+1}}{\Delta^2} - \rho \tag{19.5.35}$$

(参考方程(19.3.16))这里我们删掉了空间下标$(j,l)$。用矩阵标记，方程(19.5.35)为

$$(\mathbf{L}_x + r\mathbf{1}) \cdot \mathbf{u}^{n+1/2} = (r\mathbf{1} - \mathbf{L}_y) \cdot \mathbf{u}^n - \Delta^2\rho \tag{19.5.36}$$

$$(\mathbf{L}_y + r\mathbf{1}) \cdot \mathbf{u}^{n+1} = (r\mathbf{1} - \mathbf{L}_x) \cdot \mathbf{u}^{n+1/2} - \Delta^2\rho \tag{19.5.37}$$

其中

$$r = \frac{2\Delta^2}{\Delta t} \tag{19.5.38}$$

方程(19.5.36)和(19.5.37)左端的矩阵是三对角型矩阵(通常还是正定的)，因而它们可以通过标准三对角算法来求解。给定 $\mathbf{u}^n$，解(19.5.36)求解 $\mathbf{u}^{n+1/2}$，代入(19.5.37)的右端则可解出 $\mathbf{u}^{n+1}$。关键的问题在于如何选择迭代参数 $r$，这与初值问题中时间步的选择类似。

和通常一样，目的是使迭代矩阵的谱半径最小。虽然详细的介绍已超出我们的范围，但可以证明，$r$ 的最佳选取会使 ADI 方法与 SOR 有相同的收敛速度。ADI 方法中，各个迭代步长比 SOR 中的要复杂得多，因此 ADI 方法显得要差一些。我们可以对每个迭代步都选同样的参数 $r$。若每步都选不同的 $r$ 也是可能的。若能按最佳方式实现，则 ADI 比 SOR 通常具有更高的效率。详细内容请参阅文献[1-4]。

这里没有完全采用 ADI 的一个原因是，在大部分应用中，它已经被下节介绍的多网格法所替代。我们建议，对普通问题(比如 20×20 )或只解一次的较大问题可采用 SOR 方法。对于后一种情形，程序设计的容易程度比机时的耗费更重要。第2.7节中的稀疏矩阵方法，偶尔也用来直接求解一组差分方程。但是对于大椭圆问题的生成解，现在几乎都选用多网格方法。

参考文献和进一步读物：


Young, D. M. 1971, *Iterative Solution of Large Linear Systems* (New York: Academic Press). [1]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §§8.3~8.6. [2]

Varga, R. S. 1962, *Matrix Iterative Analysis* (Englewood Cliffs, NJ: Prentice-Hall). [3]

Spanier, J. 1967, in *Mathematical Methods for Digital Computers*, *Volume 2* (New York: Wiley), Chapter 11. [4]


## 19.6 边界值问题的多网格法

实用多网格法是 Brandt 在七十年代首次提出的。这些方法可以用 $O(N)$ 次运算，求解

在 $N$ 个网格点上的椭圆型偏微分方程。第19.4节中，讨论过的"快速"直接椭圆求解法，可用 $O(N\log N)$ 次运算，求解一些特殊类型的椭圆方程。这些估计值中的数值系数，使得多网格方法在运行速度上可以和快速方法相比较。但是与快速方法不同的是，多网格方法求解非常系数的一般椭圆型方程的效率几乎没有下降。甚至对非线性方程也有类似的求解速度。

遗憾的是，没有一种单独的多网格算法，能解所有椭圆型问题。多网格方法只是对这些问题的求解提供了一个框架，对于特定问题，必须对算法框架内的各种组成进行调整，我们在这里只能给出关于这专题的简单介绍。特别是我们将给出两个多网格程序范例，一个是线性，一个是非线性的。仿照这些原型并参阅文献[1~4]，读者能够学会编写程序，来求解自己的问题。

多网格技术的应用有两种相互联系、但又明显不同的方法。第一种称为"多网格法"，是用来提高传统松驰法的收敛速度的一种方法，由读者自己定义一个事先确定的精细网格。在这种情形，只需在该网格上定义自己的问题(例如，估计源项)。而其它一些较粗的，由方法定义的网格可以作为中间计算的附加值。

第二种方法称为(也许容易混淆)"完全多网格(FMG)法"，要求读者将同一个基本偏微分方程离散成不同尺度的有限差分方程组)。在该方法中，随着网格的不断加细得到一系列解。读者可以停在一个预先确定的精细的网格上，也可以监视由离散过程引起的截断误差，使其停在误差可以容忍的网格上。

本节我们将先讨论"多网格法"，然后利用刚才建立的概念介绍 FMG 法。我们在附加程序中，将实施后一种算法。

### 19.6.1 从一网格，到两网格，再到多网格

多网格法的关键思想，可以通过研究最简单的两网格法来理解。假定我们想解线性椭圆问题

$$\mathscr{L}u = f \tag{19.6.1}$$

其中 $\mathscr{L}$ 是某个线性椭圆算子，$f$ 是源项。在一个网孔大小为 $h$ 的均匀网格上，离散化方程(19.6.1)，得到的线性代数方程组为

$$\mathscr{L}_h u_h = f_h \tag{19.6.2}$$

令 $\tilde{u}_h$ 表示方程(19.6.2)的某近似解。用符号 $u_h$ 表示差分方程(19.6.2)的精确解。则 $\tilde{u}_h$ 的**误差**或**修正值**为

$$v_h = u_h - \tilde{u}_h \tag{19.6.3}$$

**残差**或**亏损**为

$$d_h = \mathscr{L}_h \tilde{u}_h - f_h \tag{19.6.4}$$

(注意，有些作者将残差定义为负亏损，式(19.6.4)究竟定义这两个量中的哪一个，还没有取得一致的看法)。由于 $\mathscr{L}_h$ 是线性的，因此误差满足

$$\mathscr{L}_h v_h = - d_h \tag{19.6.5}$$

现在，我们需要用 $\mathscr{L}_h$ 的一个近似值来找 $v_h$；经典迭代方法，例如雅可比或高斯-赛德尔法，是通过每步寻找下述方程的一个近似解来完成的。

$$\hat{\mathscr{L}}_h \hat{v}_h = - d_h \tag{19.6.6}$$

其中 $\hat{\mathscr{L}}_h$ 是一个比 $\mathscr{L}_h$"较简单"的算子。例如，在雅可比迭代中，$\hat{\mathscr{L}}_h$ 是 $\mathscr{L}_h$ 的对角部分，或在

高斯–赛德尔迭代中，$\hat{\mathscr{L}}_h$ 是 $\mathscr{L}_h$ 的下三角部分。第二个近似是根据下式得到的

$$\tilde{u}_h^{新} = \tilde{u}_h + \hat{v}_h \tag{19.6.7}$$

作为另一种选择，现在再考虑对 $\mathscr{L}_h$ 的一种完全不同的近似。其中我们使它"粗糙化"，而不是"简单化"。即，我们在网孔大小为 $H$（常取 $H=2h$，但别的取法也可以）的一个更粗糙的网格上，构造 $\mathscr{L}_h$ 的某个适当近似算子 $\mathscr{L}_H$。则残差方程(19.6.5)近似为

$$\mathscr{L}_H v_H = - d_H \tag{19.6.8}$$

由于 $\mathscr{L}_H$ 维数较小，该方程将比方程(19.6.5)易解。为了定义粗网格上的亏损 $d_H$，我们需要一个**限制算子** $\mathscr{R}$，将 $d_h$ 限制在粗网格上

$$d_H = \mathscr{R} d_h \tag{19.6.9}$$

限制算子也称为**细—粗算子或嵌入算子**。一旦得到方程(19.6.8)的一个解 $\tilde{v}_H$ 后，我们还需要一个**拓展算子** $\mathscr{P}$，它将修正值拓展或内插到细网格上：

$$\tilde{v}_h = \mathscr{P} \tilde{u}_H \tag{19.6.10}$$

拓展算子也称为**粗—细算子或内插算子**。$\mathscr{R}$ 和 $\mathscr{P}$ 都取为线性算子。最后 $\tilde{u}_h$ 近似值可以校正为

$$\tilde{u}^{新}_h = \tilde{u}_h + \tilde{v}_h \tag{19.6.11}$$

因此，**粗网格修正格式**的步骤是：

**粗网格修正**

- 根据式(19.6.4)计算细网格点上的亏损。
- 通过式(19.6.9)来限制亏损。
- 在粗网格上解式(19.6.8)，求修正值。
- 通过式(19.6.10)将修正值内插到细网格上。
- 通过式(19.6.11)计算第二个近似。

让我们来比较一下松驰法和粗网格修正格式的优劣。考虑将误差 $v_h$ 展开成一个离散傅里叶序列，称频率谱低半部分的分量为**光滑分量**，而高半部分为**非光滑分量**。我们已经看到，松驰法在极限 $h\to 0$ 时的情形，收敛速度变得很慢，这相当于网格点数目很多的情形，原因是，在每次迭代中光滑分量的幅度只稍微减小一点。然而，许多松驰法却使非光滑分量的幅度，在每次迭代中都减少许多倍；所以，它们是很好的**光滑算子**。

另一方面，对于两网格迭代法来说，波长 $\lesssim 2H$ 的误差分量，在粗网格点上甚至不能表示出来，因而在该网格上不能降为零。但这些高频成分却可以通过松驰法在细网格上被减小；这就启发我们，将松驰法和粗网格修正的思想结合起来使用。

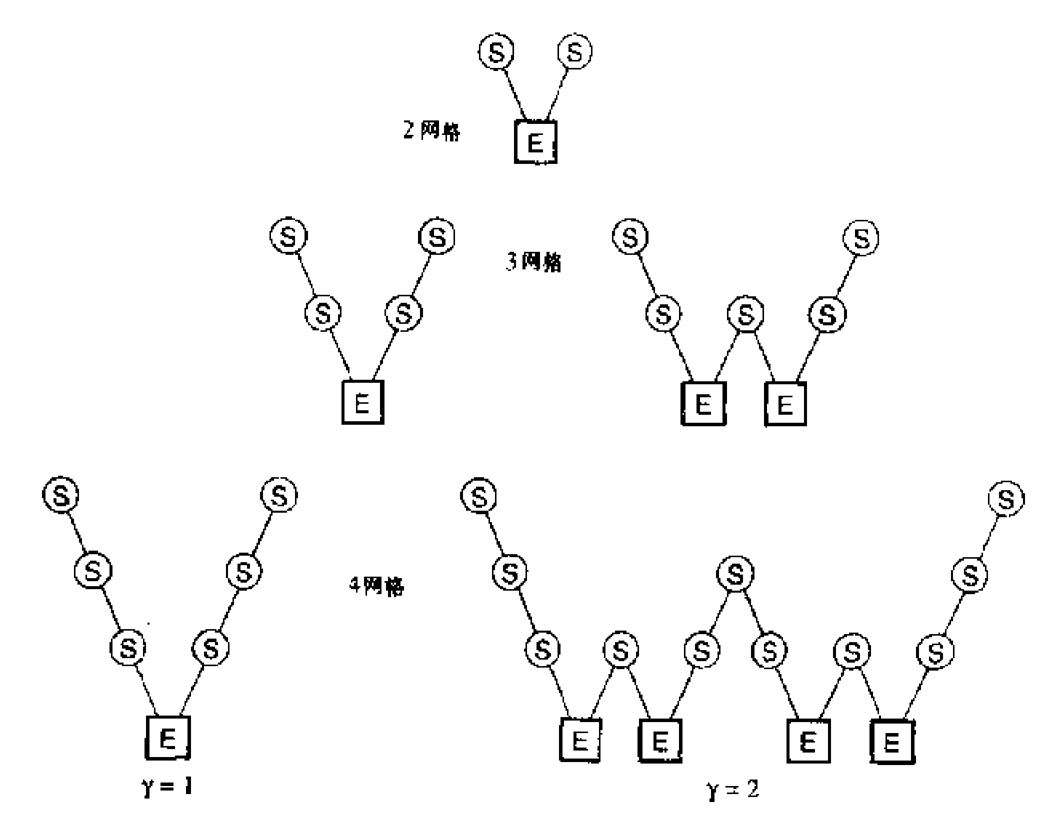
**两网格迭代**

- 预先光滑：将松驰法中 $\nu_1\geqslant 0$ 的步骤应用到 $\tilde{u}_h$ 上来计算 $\bar{u}_h$。
- 粗网格修正：和前文一样，利用 $\bar{u}_h$ 给出 $\bar{u}_h^{新}$。
- 后期光滑：将松驰法中 $\nu_2\geqslant 0$ 的步骤应用到 $\bar{u}_h^{新}$ 上，来计算出 $\tilde{\bar{u}}_h^{新}$。

从上述两网格法到多网格法就很容易。不用求解粗网格亏损方程(19.6.8)，而我们可以通过引进一个更粗的网格，并应用两网格迭代法，得到它的一个近似解。如果两网格法的收

敛因子足够小，则我们将只需几步迭代就能得到解的一个足够的近似值。我们用 $\gamma$ 来表示这样的迭代次数。显然，我们可能将该思想递归地用到一些很粗的网格上。在那网格上求解很容易，例如，通过直接矩阵求逆的方法，或通过迭代松驰格式到收敛值。

多网格法中，从最细网格到较粗网格，再回到最细网格的一次迭代称为一个循环。循环的精确结构依赖于 $\gamma$ 值，而 $\gamma$ 值是，在每个中间步骤，两网格法迭代的次数。$\gamma=1$ 的情形，称为 V 循环；而 $\gamma=2$ 的情形，称为 W 循环（见图19.6.1）。这些是实际应用中最重要的情形。




S 表示光滑度，E 表示最粗网格上的精确解，每个下降线\表示限制($\mathscr{R}$)，每个上升线/表示拓展($\mathscr{P}$)。最细的网格在每幅图的顶部。对于 V 循环($\gamma=1$)，每当网格层数增加1时，E 步就被一个双网格迭代所代替。对于 W 循环($\gamma=2$)，E 步被两个两网格迭代所代替。

图19.6.1 多网格循环结构

注意，一旦涉及两个以上的网格时，在最细网格上，第一步之后的预先光滑步，需要误差 $v$ 初始近似值，该值应取为零。

## 19.6.2 光滑，限制及拓展算子

最流行，而且也是应该最先试验的光滑方法是高斯-赛德尔(Gauss-Seidel)法，因为它能产生很好的收敛速度。如果我们将网格点从1顺序编号到 $N$，则高斯-赛德尔格式为

$$u_i = -\left(\sum_{\substack{j=1\\j\neq i}}^{N} L_{ij}u_j - f_i\right)\frac{1}{L_{ii}} \qquad i=1,\ldots,N \qquad (19.6.12)$$

其中 $u$ 的新值放在右端项使用。高斯-赛德尔法的精确形式依赖于网格点所选取的排序。对于我们典型方程(19.0.3)的典型二次椭圆方程，和方程(19.0.8)的差分格式一样，最好用红

一黑排序，使一个通过网格修正“偶”点(像一个方格盘上的红格)，而使另一个通过网格修正“奇”点。当沿某一维比沿另一维的耦合程度更强时，就需要沿该维同时将整条线松驰。最邻近耦合的线松驰涉及一个三对角系统的求解，因而仍然具有很高的效率。相继对奇偶线进行的松驰过程称为斑状松驰，通常比简单的线松驰更有效。

注意，SOR 不能用做光滑算子。超松驰会破坏高频光滑成分，这对多网格法是至关重要的。

拓展和限制算子的一个简洁的表示方法是，给出它们的**记号**。可以通过考虑 $v_H$ 在某些网格点$(x,y)$上取1，在其外点上取 0，再定义 $\mathscr{P}v_H$，得到记号 $\mathscr{P}$。最流行的拓展算子是简单的双线性插值。它给出 9 个点$(x,y)$，$(x+h,y)$，…$(x-h,y-h)$上的非零值，分别为 1，1/2，…，1/4 。因此它的记号为

$$\begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 1 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{bmatrix} \tag{19.6.13}$$

通过考虑在细网格的各点上定义 $v_h$ 再将$(x,y)$处的 $\mathscr{R}v_h$ 定义为这些值的一个线性组合，然后就得到记号 $\mathscr{R}$ 的定义。$\mathscr{R}$ 最简单的一种可能选法是**直接嵌入**，即在每个粗网格点中简单地用对应的细网格点的值来填充，它的记号是“[1]”。但是，这种取法在实际应用中可能会产生困难。可以证明 $\mathscr{R}$ 的安全选法是，使它为 $\mathscr{P}$ 的一个附加算子。为了定义附加算子，首先将网格尺度为 $h$ 的两个网格函数 $u_h$ 和 $v_h$ 的标量积，定义为

$$\langle u_h|v_h\rangle_h \equiv h^2\sum_{x,y}u_h(x,y)v_h(x,y) \tag{19.6.14}$$

然后，将 $\mathscr{P}$ 的附加算子，用 $\mathscr{P}^{\dagger}$ 表示，定义为

$$\langle u_H|\mathscr{P}^{\dagger}v_h\rangle_H = \langle \mathscr{P}u_H|v_h\rangle_h \tag{19.6.15}$$

现在，取 $\mathscr{P}$ 为双线性插值，并选定$(x,y)$处的 $u_H=1$，其余各处为零。令式(19.6.15)中的 $\mathscr{P}^{\dagger}=\mathscr{R}$，$H=2h$。可以看出

$$(\mathscr{R}v_h)_{(x,y)} = \frac{1}{4}v_h(x,y) + \frac{1}{8}v_h(x+h,y) + \frac{1}{16}v_h(x+h,y+h) + \cdots \tag{19.6.16}$$

因此 $\mathscr{R}$ 的记号为

$$\begin{bmatrix} \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \end{bmatrix} \tag{19.6.17}$$

注意下述的简单法则：$\mathscr{R}$ 记号是 $\mathscr{P}$ 记号定义的矩阵(19.6.13)乘以1/4后的变换矩阵。只要 $\mathscr{R}=\mathscr{P}^{\dagger}$ 且 $H=2h$，该法则总成立。

(19.6.17)中的 $\mathscr{R}$ 的特定选取称为**完全加权**。$\mathscr{R}$ 的另一个常用选法是**半加权**，是完全加权和直接嵌入之间的“折衷”。它的记号是

$$\begin{bmatrix} 0 & \frac{1}{8} & 0 \\ \frac{1}{8} & \frac{1}{2} & \frac{1}{8} \\ 0 & \frac{1}{8} & 0 \end{bmatrix} \tag{19.6.18}$$

差分算子 $\mathscr{L}_h$ 可用类似的记号来描述。例如，典型问题，方程(19.0.6)的标准差分由**五点差分星形**来代表

$$\mathscr{L}_h = \frac{1}{h^2}\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{19.6.19}$$

如果遇到一个新问题，而且不能确定选取怎样的 $\mathscr{P}$ 和 $\mathscr{R}$ 才能取得较好的效果时，下面是一个保险的法则：假定 $m_p$ 是插值 $\mathscr{P}$ 的阶(即，它对 $m_p-1$次多项式精确插值)，假定 $m_r$ 是 $\mathscr{R}$ 的阶，$\mathscr{R}$ 是某个 $\mathscr{P}$ 的附加算子(不一定是想用的 $\mathscr{P}$)。那么，如果 $m$ 是差分算子 $\mathscr{L}_h$ 的阶，则应当满足不等式 $m_p+m_r>m$。例如泊松方程的双线性插值及其附加算子、完全加权，满足 $m_p+m_r=4>m=2$ 。

当然 $\mathscr{P}$ 和 $\mathscr{R}$ 算子应该将边界条件施加到要求的问题中。最简单的方法是，将差分方程改写为具有齐次边界条件的形式，如果需要的话，这可以通过改变源项来实现(参看第19.4节)。施加齐次边界条件，只要求 $\mathscr{P}$ 算子在适当的边界点简单地产生零值即可。相应的 $\mathscr{R}$ 可以根据 $\mathscr{R}=\mathscr{P}^+$ 得到。

### 19.6.3 完全多网格算法

迄今为止，我们所描述的多网格法都是一种迭代格式，从最细网格上的一些初始猜测值开始，经过足够的循环(V 循环，W 循环，…)后，达到收敛。这是多网格最简单的一种用法：简单地应用足够多次循环，直到符合某个适当的收敛标准为止。但利用**完全网格**算法(FMG)可以提高效率，FMG 也称为**嵌套迭代**。

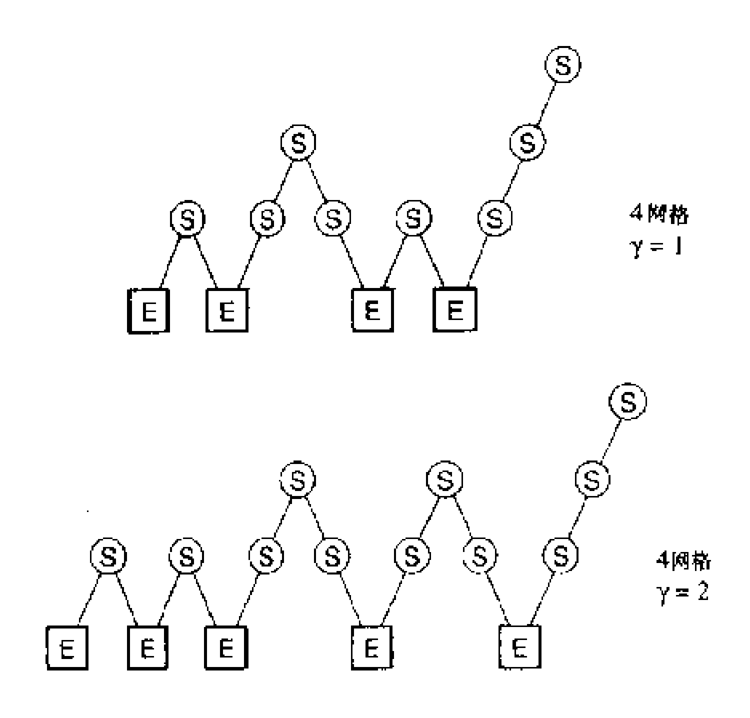
不从最细网格上的任意近似值开始(例如，$u_h=0$)，而第一个近似值是通过对一个粗网格解的内插得到：

$$u_h = \mathscr{P}u_H \tag{19.6.20}$$

粗网格解是利用类似 FMG 的过程，从更粗网格上得到的。在最粗的网格上，用精确值开始。和图19.6.1中的过程不同，FMG 是通过一系列越来越高的“$N$”到达解的，每个较高的位置都表示在较细的网格上得出的解。(见图19.6.2)。

注意，式(19.6.20)中的 $\mathscr{P}$，不一定和多网格循环中的 $\mathscr{P}$ 相同。它应该至少与离散化 $\mathscr{L}_h$ 同阶，但有时一个高阶算子可能会有更高的效率。

可以证明，在进入下个更细网格之前，在每一层次上通常需要一次或最多两次的多网格循环。虽然，所需循环次数有理论上的指导(例如[2])，但人们可以根据经验很容易地确定它。先固定最细的网格，然后增加每一层网格上的循环次数，并研究解值的变化。这些解的渐近值就是差分方程的精确值。该精确解与小数目循环次数得到的解之差为迭代误差。现在，再固定循环次数为某个较大的值，改变网格种类的数目，即所使用的最小 $h$ 值。用这种方法可以估算一个给定 h 的截断误差。最终得到的结论是，用多次循环毫无意义，并不能使迭代误差降低到截断误差的大小水平。




该方法从最粗的网格开始,先插值,然后加细(通过"V"),得到不断变细的网格上的解。

图19.6.2　完全多网格(FMG)法的循环结构

简单多网格迭代(循环),只在最细网格上需要用右端项 f。而 FMG 则在所有网格上都需要 $f$。如果边界条件是齐次的,则可以使用 $f_H=\mathscr{R}f_h$。对于非齐次边界条件,该等式不总是成立的。此时,最好在每种粗网格上都将 $f$ 离散。

注意,FMG 算法能求得所有网格上的解。因而它可以和理查德森外推法之类的技巧一起使用。

现在,我们给出一个程序 mglin,它利用完全多网格算法来解一个线性方程,如典型的问题(19.0.6)。它用红-黑高斯-赛德尔作为光滑算子,取 $\mathscr{P}$ 为双线性插值,$\mathscr{R}$ 为半加权。为使程序能处理其它线性问题,只需适当修改函数 **relax**、**resid** 以及 **slvsml** 即可。该程序的一个特点是,定义在不同网格上的变量采用动态存储。

```
#include "nrutil.h"
#define NPRE 1                     在…之前松驰扫描的次数
#define NPOST 1                    粗网格修正计算后
#define NGMAX 15

void mglin(double **u, int n, int ncycle)
```

完全多网格算法用来解线性椭圆方程,这里为典型问题(19.0.6)。输入 u[1..n][1..n]包含右端 $\rho$,而输出则返回解。维数 n 必须是 $2^j+1$,$j$ 为某个整数($j$ 实际上是求解中用到的网格层数,程序中称为 ng)。ncycle 是每层用到的 V 循环数。

```
{
    void addint(double **uf, double **uc, double **res, int nf);
    void copy(double **aout, double **ain, int n);
    void fill0(double **u, int n);
    void interp(double **uf, double **uc, int nf);
    void relax(double **u, double **rhs, int n);
    void resid(double **res, double **u, double **rhs, int n);
```

```
    void rstrct(double **uc, double **uf, int nc);
    void slvsml(double **u, double **rhs);
    unsigned int j,jcycle,jj,jpost,jpre,nf,ng=0,ngrid,nn;
    double **ires[NGMAX+1],**irho[NGMAX+1],**irhs[NGMAX+1],**iu[NGMAX+1];

    nn=n;
    while (nn >>= 1) ng++;
    if (n != 1+(1L << ng)) nrerror("n-1 must be a power of 2 in mglin.");
    if (ng > NGMAX) nrerror("increase NGMAX in mglin.");
    nn=n/2+1;
    ngrid=ng-1;
    irho[ngrid]=dmatrix(1,nn,1,nn);          给网格ng-1上的r.h.s分配存储地址，并通
    rstrct(irho[ngrid],u,nn);                 过细网格上的限制值将其填写。类似地给所
    while (nn > 3) {                          有粗网格上的r.h.s分配地址并填写
        nn=nn/2+1;
        irho[--ngrid]=dmatrix(1,nn,1,nn);
        rstrct(irho[ngrid],irho[ngrid+1],nn);
    }
    nn=3;
    iu[1]=dmatrix(1,nn,1,nn);
    irhs[1]=dmatrix(1,nn,1,nn);
    slvsml(iu[1],irho[1]);                    最粗网格上的初始解
    free_dmatrix(irho[1],1,nn,1,nn);
    ngrid=ng;
    for (j=2;j<=ngrid;j++) {                  嵌套迭代循环
        nn=2*nn-1;
        iu[j]=dmatrix(1,nn,1,nn);
        irhs[j]=dmatrix(1,nn,1,nn);
        ires[j]=dmatrix(1,nn,1,nn);
        interp(iu[j],iu[j-1],nn);                       从粗网格到下一个细网格作插值

        copy(irhs[j],(j != ngrid ? irho[j] : u),nn);    建立r.h.r的V循环
        for (jcycle=1;jcycle<=ncycle;jcycle++) {
            nf=nn;
            for (jj=j;jj>=2;jj--) {                     V的向下加添
            for (jpre=1;jpre<=NPRE;jpre++)              预光滑处理
                relax(iu[jj],irhs[jj],nf);
            resid(ires[jj],iu[jj],irhs[jj],nf);
            nf=nf/2+1;
            rstrct(irhs[jj-1],ires[jj],nf); 残差的限制是下一步r.h.s

            fill0(iu[jj-1],nf);                         将下一步松弛中的初始猜测值
            }                                               置为0
            slvsml(iu[1],irhs[1]);                      V的底端：在最粗网格
            nf=3;                                           上的求解
            for (jj=2;jj<=j;jj++) {                     V的向上加添!
            nf=2*nf-1;
            addint(iu[jj],iu[jj-1],ires[jj],nf); 在addint中res用作暂时存储空间

            for (jpost=1;jpost<=NPOST;jpost++)          后期光滑处理
                relax(iu[jj],irhs[jj],nf);
            }
        }
    }
    copy(u,iu[ngrid],n);                        返回u中的解
    for (nn=n,j=ng;j>=2;j--,nn=nn/2+1) {
        free_dmatrix(ires[j],1,nn,1,nn);
        free_dmatrix(irhs[j],1,nn,1,nn);
        free_dmatrix(iu[j],1,nn,1,nn);
        if (j != ng) free_dmatrix(irho[j],1,nn,1,nn);
    }
    free_dmatrix(irhs[1],1,3,1,3);
    free_dmatrix(iu[1],1,3,1,3);
}

void rstrct(double **uc, double **uf, int nc)
```

半加权限制，nc 是粗网格的维数。在 uf[1..2*nc-1][1..2*nc-1]中输入是细网格解，粗网格解返回到 uc[1..nc][1..nc]中。

```
{
    int ic,iif,jc,jf,ncc=2*nc-1;

    for (jf=3,jc=2;jc<nc;jc++,jf+=2) {                         内部点
        for (iif=3,ic=2;ic<nc;ic++,iif+=2) {
            uc[ic][jc]=0.5*uf[iif][jf]+0.125*(uf[iif+1][jf]+uf[iif-1][jf]
                        +uf[iif][jf+1]+uf[iif][jf-1]);
        }
    }
    for (jc=1,ic=1;ic<=nc;ic++,jc+=2) {                         边界点
        uc[ic][1]=uf[jc][1];
        uc[ic][nc]=uf[jc][ncc];
    }
    for (jc=1,ic=1;ic<=nc;ic++,jc+=2) {
        uc[1][ic]=uf[1][jc];
        uc[nc][ic]=uf[ncc][jc];
    }
}
```

```
void interp(double **uf, double **uc, int nf)
```

通过双线性插值得到粗-细拓展，nf 是细网格的维数，uc[1..nc][1..nc]作为粗网格解的输入，其中 nc=nf/2+1，细网格解返回到 uf[1..nf][1..nf]。

```
{
    int ic, iif, jc, jf, nc;
    nc=nf/2+1;
    for (jc=1,jf=1;jc<=nc; jc++,jf+=2)                          复制循环
        for (ic=1;ic<=nc;ic++) uf[2*ic-1][jf]=uc[ic][jc];
    for (jf=1;jf<=nf;jf+=2)                                    奇数列循环，竖直插值
        for (iif=2;iif<nf;iif+=2)
            uf[iif][jf]=0.5*(uf[iif+1][jf]+uf[iif-1][jf]);

    for (jf=2;jf<nf;jf+=2)                                     偶数列循环，水平插值
        for (iif=1;iif <= nf; iif++)
            uf[iif][jf]=0.5*(uf[iif][jf+1]+uf[iif][jf-1]);
}
```

```
void addint(double **uf, double **uc, double **res, int nf)
```

进行粗-细插值并将结果放在 uf 中，nf 是细网格的维数。在 uc[1..nc][1..nc]中输入是粗网格解，其中 nc=nf/2+1，细网格解返回到 uf[1..nf][1..nf]中，res[1..nf][1..nf]作为暂时存储空间。

```
{
    void interp(double **uf, double **uc, int nf);
    int i,j;

    interp(res, uc,nf);
    for (j=1;j<=nf;j++)
        for (i=1;i<=nf;i++)
            uf[i][j] += res[i][j];
}
```

```
void slvsml(double **u, double **rhs)
```

在最粗网格上典型问题的求解，其中 h=1/2，在 rhs[1..3][1..3]中是右端项的输入，解返回到 u[1..3][1..3] 中。

```
{
    void fill0(double **u, int n);
    double h=0.5;

    fill0(u,3);
```

```
    u[2][2] = -h*h*rhs[2][2]/4.0;
}

void relax(double **u, double **rhs, int n)
    典型问题的红一黑高斯-赛得尔松驰,利用右端项函数 rhs[1..n][1..n],修正解 u[1..n][1..n]的当前值
{
    int i,ipass,isw,j,jsw=1;
    double h,h2;

    h=1.0/(n-1);
    h2=h*h;
    for (ipass=1;ipass<=2;ipass++,jsw=3-jsw) {                    红和黑扫描
        isw=jsw;
        for (j=2;j<n;j++,isw=3-isw)
            for (i=isw+1;i<n;i+=2)                                Gass Seidel 公式
                u[i][j]=0.25*(u[i+1][j]+u[i-1][j]+u[i][j+1]
                        +u[i][j-1]-h2*rhs[i][j]);
    }
}

void resid(double **res, double **u, double **rhs, int n)
    对典型问题返回负残差,输入量是 u[1..n][1..n],和 rhs[1..n][1..n]而返回到 res[1..n][1..n].
{
    int i,j;
    double h,h2i;

    h=1.0/(n-1);
    h2i=1.0/(h*h);
    for (j=2;j<n;j++)                                             内部点
        for (i=2;i<n;i++)
            res[i][j] = -h2i*(u[i+1][j]+u[i-1][j]+u[i][j+1]+u[i][j-1]-
                    4.0*u[i][j])+rhs[i][j];
    for (i=1;i<=n;i++)                                            边界点
        res[i][1]=res[i][n]=res[1][i]=res[n][i]=0.0;
}

void copy(double **aout, double **ain, int n)          将 ain[1..n][1..n]复制到 aout[1..n][1..n]中
{
    int i,j;
    for (i=1;i<=n;i++)
        for (j=1;j<=n;j++)
            aout[j][i]=ain[j][i];

}

void fill0(double **u, int n)      将 u[1..n][1..n]中零元填充。
{
    int i,j;
    for (j=1;j<=n;j++)
        for (i=1;i<=n;i++)
            u[i][j]=0.0;
}
```

程序 **mglin** 的编写是为了使算法更加明确,而不是为了追求最高的效率,因而很容易对它进行改进。下面几个简单的修改将会使运行时间缩短。

- 红一黑高斯一赛德尔步后,所有黑网格点上的亏损 $d_h$ 全部为零。因此,半加权 $d_H = \mathcal{R} d_h$ 变为只需简单地将细网格上亏损的一半复制到相应的粗网格上。在 V 循环的第

一部分中 **rstrct** 前对 **resid** 的调用可以通过一个只在粗网格上循环的程序来代替。它在粗网格上填进一半亏损。

- 类似地，红网格点上的量 $\tilde{u}_h^{new}=\tilde{u}_h+\mathscr{P}\tilde{v}_H$ 不用计算，因为它们在接着进行的高斯-赛德尔扫描中，将立即被重新定义。这意味着 **addint** 只需在黑网点上进行循环。
- 可以用多种方法使 **relax** 提高运行速度。首先，当初始猜测值取零时，可以得到一个特殊的形式并省略程序 **fillo**。其次，可以将各种网格上的 $h^2f_h$ 存起来，并存入一个倍数。最后，可以通过中间变量重写高斯-赛德尔公式，将公式中的一个加法存起来。
- 对于典型问题，ncyclu＝1时，对于给定 $h$ 的大小，由 **mglin** 返回的解其迭代误差将大于截断误差。为了将误差降低到截断误差的水平，必须置 ncycle＝2，或更经济地置 npre＝2 。一个更有效的方法是，在方程(19.6.20)中采用一个更高阶的 $\mathscr{P}$，它比 V 循环中使用的线性插值阶次更高。

实现所有上述特性，通常可以使运行时间缩短到原来的 1/2，并且最终结果当然很有用。

### 19.6.4 非线性多网格：FAS 算法

现在，再来求解非线性椭圆方程，我们将其形式上写为

$$\mathscr{L}(u)=0 \tag{19.6.21}$$

任意显含的源项都移到了左端。假定方程(19.6.21)被适当离散为：

$$\mathscr{L}_h(u_h)=0 \tag{19.6.22}$$

下面，将会看到在多网格算法中，我们必须考虑求解过程中，产生非零右端项的方程：

$$\mathscr{L}_h(u_h)=f_h \tag{19.6.23}$$

用多网格解非线性问题的一个途径是，利用牛顿法；即在每步迭代时，对修正项产生线性方程，于是，我们可以利用线性多网格来解这些方程。然而，多网格思想的巨大优势，表现在它可以直接应用到非线性问题。我们要做的只是，选一个适当的**非线性**松驰法来光滑误差，再加上一个估算粗网格上近似修正值的过程。

这个直接方法就是，布拉德(Brandt)的完全近似存储算法(FAS)。也许除最粗网格外，一般没有非线性方程需要求解。

这为实现非线性算法，假定我们象线性情形一样，已经有一个能光滑残差向量的松驰法过程。于是我们可以找一个光滑的修正值 $v_h$ 来解方程(19.6.23)：

$$\mathscr{L}_h(\tilde{u}_h+v_h)=f_h \tag{19.6.24}$$

为了找 $v_h$，注意

$$\mathscr{L}_h(\tilde{u}_h+v_h)-\mathscr{L}_h(\tilde{u}_h)=f_h-\mathscr{L}_h(\tilde{u}_h)=-d_h \tag{19.6.25}$$

经过几次非线性松驰扫描后，右端是光滑的。因而，我们可以将左端转移到一个粗网格上：

$$\mathscr{L}_H(u_H)-\mathscr{L}_H(\mathscr{R}\tilde{u}_h)=-\mathscr{R}d_h \tag{19.6.26}$$

即，我们在粗网格上解

$$\mathscr{L}_H(u_H)=\mathscr{L}_H(\mathscr{R}\tilde{u}_h)-\mathscr{R}d_h \tag{19.6.27}$$

(非零右端项就是这样出现的)。假定近似解为 $\tilde{u}_H$，则粗网格上的修正值为

$$\tilde{v}_H=\tilde{u}_H-\mathscr{R}\tilde{u}_h \tag{19.6.28}$$

以及

$$\tilde{u}_h^{new}=\tilde{u}_h+\mathscr{P}(\tilde{u}_H-\mathscr{R}\tilde{u}_h) \tag{19.6.29}$$

注意，一般情况下 $\mathscr{R}\mathscr{P}\neq 1$，因此 $\tilde{u}_h^{\text{new}}\neq\mathscr{P}\tilde{u}_H$。这是关键点：方程(19.6.29)中的插值误差只由修正值产生，而不是由整个 $\mathscr{P}\tilde{u}_H$ 产生。

方程(19.6.27)表明，我们求解的是完全近似值 $u_H$，而不是线性算法中只求误差，这就是 FAS 名称的由来。

因此，FAS 多网格算法与线性多网格算法看来极其相似。仅有差别是，亏损 $d_h$ 和松驰近似值 $\tilde{u}_h$ 都必须被限制在粗网格上，通过对算法的递归调用，求解方程(19.6.27)。但是，不通过这种途径也可实现算法，我们将先介绍所谓的**对偶观点**，它能使我们看到多网格思想的另一个方面优势。

对偶观点考虑**局部截断误差**，定义为

$$\tau\equiv\mathscr{L}_h(u)-f_h \tag{19.6.30}$$

其中 $u$ 是原始连续介质方程的精确解。若将上式改写为

$$\mathscr{L}_h(u)=f_h+\tau \tag{19.6.31}$$

我们看到 $\tau$ 可以作为对 $f_h$ 的修正值，因而细网格上方程的解将成为精确解 $u$。

现在，考虑**相对截断误差** $\tau_h$，它定义在相对于 $h$ 网格的 $H$ 网格上：

$$\tau_h\equiv\mathscr{L}_H(\mathscr{R}u_h)-\mathscr{R}\mathscr{L}_h(u_h) \tag{19.6.32}$$

由于 $\mathscr{L}_h(u_h)=f_h$，上式可改写为

$$\mathscr{L}_H(u_H)=f_H+\tau_h \tag{19.6.33}$$

换言之，我们可以认为 $f_H$ 经过 $\tau_h$ 修正后，使得粗网格方程解与细网格方程解相等。当然，我们不能计算出 $\tau_h$，但却可以利用方程(19.6.32)中的 $u_h$ 给出它的一个近似值：

$$\tau_h\simeq\tilde{\tau}_h\equiv\mathscr{L}_H(\mathscr{R}\tilde{u}_h)-\mathscr{R}\mathscr{L}_h(\tilde{u}_h) \tag{19.6.34}$$

将方程(19.6.33)中的 $\tau_h$ 用 $\tilde{\tau}_h$ 代替给出

$$\mathscr{L}_H(u_H)=\mathscr{L}_H(\mathscr{R}\tilde{u}_h)-\mathscr{R}d_h \tag{19.6.35}$$

正是粗网格方程(19.6.27)!

因此我们看到，粗细网格之间的关系有两种互补的观点：

- 粗网格用来提高细网格残差的光滑成分的收敛速度。
- 细网格用来计算对粗网格方程的修正值，使粗网格上得到细网格上的精度。

这种新观念的一个优点是，对于多网格迭代，利用它可以得到一个自然的停止标准。这个标准通常为

$$\|d_h\|<\varepsilon \tag{19.6.36}$$

问题在于如何选取 $\varepsilon$。显然，当剩余误差基本上为局部截断误差 $\tau$ 时，再进行迭代就毫无益处了。而可计算量是 $\tilde{\tau}_h$，那么 $\tau$ 和 $\tilde{\tau}_h$ 之间的关系是什么呢？对于二阶精确差分格式，这样的典型情形有

$$\tau=\mathscr{L}_h(u)-\mathscr{L}_h(u_h)=h^2\tau_2(x,y)+\cdots \tag{19.6.37}$$

假定解满足 $u_h=u+h^2u_2(x,y)+\cdots$，然后假定 $\mathscr{R}$ 的阶足够高，以致于我们可以忽略它的作用，则方程(19.6.32)给出

$$\begin{aligned}\tau_h&\simeq\mathscr{L}_H(u+h^2u_2)-\mathscr{L}_h(u+h^2u_2)\\&=\mathscr{L}_H(u)-\mathscr{L}_h(u)+h^2[\mathscr{L}_H'(u_2)-\mathscr{L}_h'(u_2)]+\cdots\\&=(H^2-h^2)\tau_2+O(h^4)\end{aligned} \tag{19.6.38}$$

对于通常情形 $H=2h$，我们可以得到

$$\tau\simeq\frac{1}{3}\tau_h\simeq\frac{1}{3}\tilde{\tau}_h \tag{19.6.39}$$

因此，停止的标准是方程(19.6.36)中

$$\varepsilon=\alpha\|\tilde{\tau}_h\|,\qquad\alpha\sim\frac{1}{3} \tag{19.6.40}$$

在实现非线性多网格算法前，我们还有一项工作：选取一个非线性松弛格式。首先可能的选择仍是非线性高斯-赛德尔格式。如果离散化方程(19.6.23)在选定排序后，写为

$$L_i(u_1,\ldots,u_N) = f_i, \quad i = 1,\ldots,N \tag{19.6.41}$$

则非线性高斯-赛德尔格式为求 $u_i^{\text{新}}$ 而解下述方程

$$L_i(u_1,\ldots,u_{i-1},u_i^{\text{新}},u_{i+1},\ldots,u_N) = f_i \tag{19.6.42}$$

和以前一样，一旦新 $u$ 计算出来就要代替旧 $u$。对于 $u_i^{\text{新}}$ 方程(19.6.42)通常是线性的，这是因为非线性项通过邻近项被离散化了。如果不是这样，我们用一步牛顿迭代来代替方程(19.6.42)

$$u_i^{\text{新}} = u_i^{\text{旧}} - \frac{L_i(u_i^{\text{旧}}) - f_i}{\partial L_i(u_i^{\text{旧}})/\partial u_i} \tag{19.6.43}$$

例如，考虑简单的非线性方程

$$\nabla^2 u - u^2 = \rho \tag{19.6.44}$$

用二维记号，我们得到

$$L(u_{i,j}) = (u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j})/h^2 + u_{i,j}^2 - \rho_{i,j} = 0 \tag{19.6.45}$$

由于

$$\frac{\partial \mathscr{L}}{\partial u_{i,j}} = -4/h^2 + 2u_{i,j} \tag{19.6.46}$$

因此，牛顿高斯-赛德尔迭代为

$$u_{i,j}^{\text{新}} = u_{i,j} - \frac{\mathscr{L}(u_{i,j})}{-4/h^2 + 2u_{i,j}} \tag{19.6.47}$$

下面是一个利用完全多网格算法及 FAS 格式求解方程(19.6.44)的程序 **mgfas**。限制和拓展与 **mglin** 中类似。收敛试验根据方程(19.6.40)来做。某个问题的一个成功的多网格解应该以满足该条件为目标，并使 V 循环数的最大值 maxcyc 等于 1 或 2 。**mgfas** 程序中，用到的 **copy**、**interp** 及 **rstrct** 函数与 **mglin** 中的函数一样。

```
#include "nrutil.h"
#define NPRE 1                在…之前的松驰扫描数
#define NPOST 1               粗网格修正值被计算后，
#define ALPHA 0.33            将截断误差的估计值与残差的范数联系起来
#define NGMAX 15

void ngfas(double **u, int n, int maxcyc)
```

用完全多网格算法求非线性椭圆方程的 FAS 解，方程如(19.6.44)。输入 u[1..n][1..n]中包含右端项 $\rho$，输出则返回解。维数 n 必须是 $2^j+1$形式，其中 $j$ 为某个整数(实际上，$j$ 是解中用到的网格层数，程序中称为 ng)。maxcyc 是每层网格中，使用 V 循环的最大次数。

```
{
    double anorm2(double **a, int n);
    void copy(double **aout, double **ain, int n);
    void interp(double **uf, double **uc, int nf);
    void lop(double **out, double **u, int n);
    void matadd(double **a, double **b, double **c, int n);
    void matsub(double **a, double **b, double **c, int n);
    void relax2(double **u, double **rhs, int n);
    void rstrct(double **uc, double **uf, int nc);
    void slvsm2(double **u, double **rhs);
    unsigned int j,jcycle,jj,jm1,jpost,jpre,nf,ng=0,ngrid,nn;
    double **irho[NGMAX+1],**irhs[NGMAX+1],**itau[NGMAX+1],
        **itemp[NGMAX+1],**iu[NGMAX+1];
    double res,trerr;

    nn=n;
```

```
while (nn >>= 1) ng++;
if (n != 1+(1L << ng)) nrerror("n-1 must be a power of 2 in mgfas.");
if (ng > NGMAX) nrerror("increase NGMAX in mglin.");
nn=n/2+1;
ngrid=ng-1;
irho[ngrid]=dmatrix(1,nn,1,nn);        给网络ng-1上的r.h.s分配存储地址，并通过
rstrct(irho[ngrid],u,nn);              细网格上的限制值将其填写，类似地给所有粗网
while (nn > 3) {                       格上的r.h.s分配存储地址并填写
    nn=nn/2+1;
    irho[--ngrid]=dmatrix(1,nn,1,nn);
    rstrct(irho[ngrid],irho[ngrid+1],nn);
}
nn=3;
iu[1]=dmatrix(1,nn,1,nn);
irhs[1]=dmatrix(1,nn,1,nn);
itau[1]=dmatrix(1,nn,1,nn);
itemp[1]=dmatrix(1,nn,1,nn);
slvsm2(iu[1],irho[1]);                 最粗网格上的初始解
free_dmatrix(irho[1],1,nn,1,nn);
ngrid=ng;
for (j=2;j<=ngrid;j++) {               嵌套迭代循环
    nn=2*nn-1;
    iu[j]=dmatrix(1,nn,1,nn);
    irhs[j]=dmatrix(1,nn,1,nn);
    itau[j]=dmatrix(1,nn,1,nn);
    itemp[j]=dmatrix(1,nn,1,nn);
    interp(iu[j],iu[j-1],nn);          从粗网格到下一个细网格作插值

    copy(irhs[j],(j != ngrid ? irho[j] : u),nn);      建立 r.h.s.
    for (jcycle=1;jcycle<=maxcyc;jcycle++) {         V循环
    nf=nn;
        for (jj=j;jj>=2;jj--) {                      V的向下添加
            for (jpre=1;jpre<=NPRE;jpre++)           预先光滑处理
                relax2(iu[jj],irhs[jj],nf);
            lop(itemp[jj],iu[jj],nf);                L_h(ũ_h).
            nf=nf/2+1;
            jm1=jj-1;
            rstrct(itemp[jm1],itemp[jj],nf);         R L_h(ũ_h).
            rstrct(iu[jm1],iu[jj],nf);               R ũ_h.
            lop(itau[jm1],iu[jm1],nf);         L_H(R ũ_h)暂时存贮在 τ_h 中

            matsub(itau[jm1],itemp[jm1],itau[jm1],nf);      构造 τ_h.
            if (jj == j)
                trerr=ALPHA*anorm2(itau[jm1],nf);    估计截止误差 τ
            rstrct(irhs[jm1],irhs[jj],nf);           f_H.
            matadd(irhs[jm1],itau[jm1],irhs[jm1],nf);    f_H + τ_h.
        }
        slvsm2(iu[1],irhs[1]);                       V的底端：在最粗网格上求解
        nf=3;
        for (jj=2;jj<=j;jj++) {                      V的向上添加
        jm1=jj-1;
        rstrct(itemp[jm1],iu[jj],nf);                R ũ_h.
        matsub(iu[jm1],itemp[jm1],itemp[jm1],nf);    ũ_H - R ũ_h.
        nf=2*nf-1;
        interp(itau[jj],itemp[jm1],nf);              P(ũ_H-R ũ_h) 存储于 τ_h.
        matadd(iu[jj],itau[jj],iu[jj],nf);           构造 ũ_h^新
        for (jpost=1;jpost<=NPOST;jpost++)           后期光滑处理
            relax2(iu[jj],irhs[jj],nf);
        }
        lop(itemp[j],iu[j],nf);                      构造残差 ||d_h||
        matsub(itemp[j],irhs[j],itemp[j],nf);
        res=anorm2(itemp[j],nf);
        if (res < trerr) break;                      如果残差足够小，就不用再
    }                                                    进行V循环
}
copy(u,iu[ngrid],n);                                 将解返回到u中
```

```
    for (nn=n,j=ng;j>=1;j--,nn=nn/2+1) {
        free_dmatrix(itemp[j],1,nn,1,nn);
        free_dmatrix(itau[j],1,nn,1,nn);
        free_dmatrix(irhs[j],1,nn,1,nn);
        free_dmatrix(iu[j],1,nn,1,nn);
        if (j != ng && j != 1) free_dmatrix(irho[j],1,nn,1,nn);
    }
}
```

```
void relax2(double **u, double **rhs, int n)
```

方程(19.6.44)的红—黑高斯-赛德尔松弛,利用右端项函数 rhs[1..n][1..n]修正解 u[1..n][1..n]的当前值。

```
{
    int i,ipass,isw,j,jsw=1;
    double foh2,h,h2i,res;

    h=1.0/(n-1);
    h2i=1.0/(h*h);
    foh2 = -4.0*h2i;
    for (ipass=1;ipass<=2;ipass++,jsw=3-jsw) {          红-黑扫描
        isw=jsw;
        for (j=2;j<n;j++,isw=3-isw) {
            for (i=isw+1;i<n;i+=2) {
                res=h2i*(u[i+1][j]+u[i-1][j]+u[i][j+1]+u[i][j-1]-
                    4.0*u[i][j])+u[i][j]*u[i][j]-rhs[i][j];
                u[i][j] -= res/(foh2+2.0*u[i][j]);       牛顿高斯-赛德尔公式
            }
        }
    }
}

#include <math.h>
```

```
void slvsm2(double **u, double **rhs)
```

方程(19.6.44)在最粗网格上的解,其中 $h=1/2$。右端项输入到 rhs[1..3][1..3],并且解返回在 u[1..3][1..3]中。

```
{
    void fill0(double **u, int n);
    double disc,fact,h=0.5;

    fill0(u,3);
    fact=2.0/(h*h);
    disc=sqrt(fact*fact+rhs[2][2]);
    u[2][2] = -rhs[2][2]/(fact+disc);
}
```

```
void lop(double **out, double **u, int n)
```

给定 u[1..n][1..n],将方程(19.6.44)的 $\mathscr{L}_h(\tilde{u}_h)$ 返回到 out[1..n][1..n]中。

```
{
    int i,j;
    double h,h2i;

    h=1.0/(n-1);
    h2i=1.0/(h*h);
    for (j=2;j<n;j++)                                       内部点
        for (i=2;i<n;i++)
            out[i][j]=h2i*(u[i+1][j]+u[i-1][j]+u[i][j+1]+u[i][j-1]-
                4.0*u[i][j])+u[i][j]*u[i][j];                边界点
    for (i=1;i<=n;i++)
        out[i][1]=out[i][n]=out[1][i]=out[n][i]=0.0;
}
```

```
void matadd(double **a, double **b, double **c, int n)
```

将 a[1..n][1..n]和 b[1..n][1..n]相加,并将结果返回到 c[1..n][1..n]中。

```
{
    int i,j;

    for (j=1;j<=n;j++)
        for (i=1;i<=n;i++)
            c[i][j]=a[i][j]+b[i][j];
}

void matsub(double **a, double **b, double **c, int n)
    从a[1..n][1..n]中减去b[1..n][1..n],并将结果返回到c[1..n][1..n]中.
{
    int i,j;

    for (j=1;j<=n;j++)
        for (i=1;i<=n;i++)
            c[i][j]=a[i][j]-b[i][j];
}

#include <math.h>

double anorm2(double **a, int n)                    返回矩阵a[1..n][1..n]的欧几里得范数
{
    int i,j;
    double sum=0.0;

    for (j=1;j<=n;j++)
        for (i=1;i<=n;i++)
            sum += a[i][j]*a[i][j];
    return sqrt(sum)/n;
}
```

## 参考文献和进一步读物:


Brandt, A. 1977, *Mathematics of Computation*, vol. 31, pp. 333~390. [1]

Hackbusch, W. 1985, *Multi-Grid Methods and Applications* (New York: Springer-Verlag). [2]

Stuben, K., and Trottenberg, U. 1982, in *Multigrid Methods*, W. Hackbusch and U. Trottenberg, eds. (Springer Lecture Notes in Mathematics No. 960) (New York: Springer Verlag), pp. 1~176. [3]

Brandt, A. 1982, in *Multigrid Methods*, W. Hackbusch and U. Trottenberg, eds. (Springer-Lecture Notes in Mathematics No. 960) (New York: Springer-Verlag). [4]

Baker, L. 1991, *More C Tools for Scientists and Engineers* (New York: McGraw Hill).

Jespersen, D. 1984, *Multrigrid Methods for Partial Differential Equations* (Washinton: Mathematical Association of America).

McCormick, S. F. (ed.) 1988, *Multigrid Methods: Theory, Applications, and Supercomputing* (New York: Marcel Dekker).

Hackbsch, W., and Trottenberg, U. (eds.) 1991, *Multigrid Methods* **I** (Boston: Birkhauser).

# 第二十章　少数的数值算法

## 20.0　引言

迄今为止，已完成了数值算法的论述，读者可以阅读到此结束。而这最后一章是一类"少数的数值算法"的特集。出于种种原因，我们还是决定把它包括在这本以多数的数值算法为主的书中。我们注意到，计算机科学教科书的作者们喜欢在他们的书中，增加一些典型的数值计算的论题（通常是一类呆板的内容——例如，求积分）。我们发现我们自己也不能回避这种倾向，在本书中插入了一些其它的内容。

我们对素材的选择并不是完全随意的。其中一个主题即格雷码，已经用于构造伪随机序列（第7.7节），在此只需作一些补充说明。另外两个主题，讨论了计算机浮点参数的诊断和任意精度的运算，它能增进对计算机的认识、即计算机更善于做数字（相对于字符和二进位）方面的工作的认识。后者这个主题，还显示了第十二章快速傅里叶变换的一个非常不同的用法。

另外三个主题（校验和式、霍夫曼码和算术码），涉及了数据编码、压缩和认证的不同方面。如果读者需处理大量数据——甚至，数值数据，那么顺便熟悉一下这些内容，可能在某一点上迟早会有帮助的。例如，在第13.6节中，我们已遇到霍夫曼编码的一个很好的运用。

但要再次说明，读者可不必非读这章内容不可（正如读者应该从第四章和第十六章中学习数值求积分，而不是从计算机科学教科书中去学习。）

## 20.1　诊断机器的参数

一个简单的假设是，计算机的浮点运算是"足够精确"的。如果相信这条假设，则数值分析就变成非常清楚的议题。不考虑舍入误差，使许多有限算法变得"精确"，所算结果和真实结果之间只差了一个截断误差（第1.3节）。这样说，听起来是不是很幼稚？

是，确实很幼稚。尽管如此，它仍是贯穿于本书大部分内容中，是我们所必须接受的假设。对于我们所讨论过的每一种算法，它们的舍入误差是如何传递或限定的?要回答好这个问题是不现实的。事实上，也是不可能的。无论是我们还是其他人都没有对许多实际算法做过严格的分析。

当听到用户说，"我遇到了单精度的舍入误差，所以我改成双精度"，这使真正的数值分析员畏缩了。这句话的实际意思是："对这个特定的算法以及某特定的数据来说，双精度似乎能恢复我认为这个简单假设是错误的这种信念"。我们承认在本书中，所提到的精度和舍入误差大部分在特性上仅仅是稍微更定量些，这是由于我们试图更"实际些"所引起的。

了解用户自己机器的浮点运算的实际限度是很重要的，当用户直觉地、实验性地、或随意地处理浮点舍入误差时尤其重要。实验性地确定有用的浮点参数的方法，已由科迪

(Cody)[2]、马尔科姆(Malclm)[3]和其他人得出、遵循科迪所提供的实施办法，可以由程序 **machar** 来具体实现。

程序 **machar** 的所有变量都是返回值，它们的含义是：

- ibeta(第1.3节中称为 $B$)是数值计数法的基数，通常为 2，偶尔为 16 或其它值。
- it 是浮点尾数 $M$ 中基数 ibeta 的位数(见图1.3.1)。
- machep 是 ibeta 的最小(通常是负的)幂次的指数值，它加 1.0 后，得到与 1.0 不同的数值。
- eps 是 $\text{ibeta}^{\text{machep}}$ 的浮点数值，它可粗略看作为"浮点精度"。
- negep 是 ibeta 的最小幂次的指数值，被 1.0 减，得到与 1.0 不同的数值
- epsneg 就是 $\text{ibeta}^{\text{negep}}$ 另一种决定浮点精度的方法，通常，epsneg 是 eps 的 0.5 倍，偶尔 eps 和 epsneg 相等。
- iexp 是指数值的位数(包括它的符号和偏置)
- mixnexp 是尾数不以零起头的 ibeta 的最小(通常是负的)值。
- xmin 是 $\text{ibeta}^{\text{minexp}}$ 的浮点数值，通常是最小可用的浮点数值。
- maxexp 是导致溢出的 ibeta 的最小(正的)值。
- max 是 $(1-\text{epsneg})\times\text{ibeta}^{\text{maxexp}}$，通常是最大可用浮点数值。
- irnd 返回一个范围从 0 到 5 的代码，它给出使用何种舍入方法以及如何处理下溢的信息，参见下文。
- ngrd 是当截断两尾数的乘积，以适应计数法时所用的"保护位"位数。

在象 **machar** 这样的程序中，有许多细微之处，对用户来说其目的是，找出明显的机器特性。另外之所以必须这样做，是为了避免象上溢和下溢这种可能中断程序执行的错误等的发生。在有些情况下，程序只有靠辨认"标准"表示法的某些特征来能，例如，它根据舍入方式来辨认 IEEE 标准表示法，并且假设该指数表示法的某些特征作为其舍入方式的结果，详细内容请参阅文献[1]及有关的参考文献。注意，在非标准机器上程序 **machar** 可能给出不正确的结果。

关于参数 irnd 还需一些附加说明。在 IEEE 标准中，二进制模式对应于精确的、可表示的数值。加法舍入的具体方法是，把两个可表示的值"精确"地加起来，然后把和数舍入成最"邻近的可表示的数值。如果这个和数精确地处在两个可表示的数中间，则应该将它舍入为偶的那个数(即低位二进制数为零)。对于所有其它的运算操作应保持同样的方法，也就是说，应把它们等同于无限精确来处理，然后舍入到最邻近的可表示的数值。

如果irnd 返回 2 或 5，则用户的计算机就是服从这一标准的。如果它返回 1 或 4，则就是采用IEEE 标准以外的舍入方法。如果它返回 0 或 3，则它就是采用截断结果而不是舍入它，一般这是不希望的情况。

由 irnd 引发的另一个问题涉及到下溢。如果一个浮点值小于 xmin，许多计算机将其值下溢至零。若 irnd 的值等于 0,1 或 2，这就说明计算机采用的是这种方式。IEEE 标准规定了一种更适度的下溢：当一个数值变得小于 xmin 时，它的指数值就固定在最小允许值上，而它的尾数下降，获得尾数起头处为零并且"适度地"丢失精度，若采用这种方式，以 irnd 等于 3,4 或 5 来指示。

```
# include <math.h>
```

```
#define CONV(i) ((float) (i))
```

此处把float型变成double型，并且在下面的说明语句中，改变相应双精度参数

```
void machar (int *ibeta, int *it, int *irnd, int *ngrd, int *machep, int *negep, int *iexp,
    int *minexp, int *maxexp, float *eps, float *epsneg, float *xmin, float *xmax)
```

确定和返回影响浮点运算的机器的特性参数，返回值包括：ibeta，浮点数的基数；it，浮点型尾数基数ibeta的位数；eps，最小正数，它加到1.0上后，不等于1.0的最小正数；epsneg，也是最小正数，它是被1.0减后，不等于1.0的最小正数；xmin，最小可表示的正数；xmax，最大可表示的正数，其它返回参数的说明参见正文。

```
{
    int i,itemp,iz,j,k,mx,nxres;
    float a,b,beta,betah,betain,one,t,temp,temp1,tempa,two,y,z,zero;

    one=CONV(1);
    two=one+one;
    zero=one-one;
    a=one;                          采用马尔科姆的方法确定ibeta和beta
    do {
        a += a;
        temp=a+one;
        temp1=temp-a;
    } while (temp1-one == zero);
    b=one;
    do {
        b += b;
        temp=a+b;
        itemp=(int)(temp-a);
    } while (itemp == 0);
    *ibeta=itemp;
    beta=CONV(*ibeta);
    *it=0;                          确定it和irnd
    b=one;
    do {
        ++(*it);
        b *= beta;
        temp=b+one;
        temp1=temp-b;
    } while (temp1-one == zero);
    *irnd=0;
    betah=beta/two;
    temp=a+betah;
    if (temp-a != zero) *irnd=1;
    tempa=a+beta;
    temp=tempa+betah;
    if (*irnd == 0 && temp-tempa != zero) *irnd=2;
    *negep=(*it)+3;                 确定negep和epsneg
    betain=one/beta;
    a=one;
    for (i=1;i<=(*negep);i++) a *= betain;
    b=a;
    for (;;) {
        temp=one-a;
        if (temp-one != zero) break;
        a *= beta;
        --(*negep);
    }
    *negep = -(*negep);
    *epsneg=a;
    *machep = -(*it)-3;             确定machep和eps
    a=b;
   for (;;) {
       temp=one+a;
       if (temp-one != zero) break;
       a *= beta;
       ++(*machep);
   }
```

```
*eps=a;
*ngrd=0;                              确定 ngrd.
temp=one+(*eps);
if (*irnd == 0 && temp*one-one != zero) *ngrd=1;
i=0;                                  确定 iexp.
k=1;
z=betain;
t=one+(*eps);
nxres=0;
for (;;) {                            循环直至下溢发生，则退出
    y=z;
    z=y*y;
    a=z*one;                          此处检查下溢发生否
    temp=z*t;
    if (a+a == zero || fabs(z) >= y) break;
    temp1=temp*betain;
    if (temp1*beta == z) break;
    ++i;
    k += k;
}
if (*ibeta != 10) {
    *iexp=i+1;
    mx=k+k;
} else {                              只用于十进制机器
    *iexp=2;
    iz=(*ibeta);
    while (k >= iz) {
        iz *= *ibeta;
        ++(*iexp);
    }
    mx=iz+iz-1;
}
for (;;) {                            为确定 minexp 和 xmin 循环直至下溢发生，
                                        则退出
    *xmin=y;
    y *= betain;
    a=y*one;                          此处检查下溢发生否
    temp=y*t;
    if (a+a != zero && fabs(y) < *xmin) {
        ++k;
        temp1=temp*betain;
        if (temp1*beta == y && temp != y) {
            nxres=3;
            *xmin=y;
            break;
        }
    }
    else break;
}
*minexp = -k;                         确定 maxexp, xmax
if (mx <= k+k-3 && *ibeta != 10) {
    mx += mx;
    ++(*iexp);
}
*maxexp=mx+(*minexp);
*irnd += nxres;                        调整 irnd 来反映部分下溢
if (*irnd >= 2) *maxexp -= 2;          为 IEEE 型机调整
i=(*maxexp)+(*minexp);
```

为在二进制尾数中具有隐含前导位的机器作调整，并且这种机型所表示的基点在尾数的最右端

```
    if (*ibeta == 2 && !i) --(*maxexp);
    if (i > 20) --(*maxexp);
    if (a != y) *maxexp -= 2;
    *xmax=one-(*epsneg);
    if ((*xmax)*one != *xmax) *xmax=one-beta*(*epsneg);
    *xmax /= (*xmin*beta*beta*beta);
    i=(*maxexp)+(*minexp)+3;
    for (j=1;j<=i;j++) {
```

```
        if (*ibeta == 2) *xmax += *xmax;
        else *xmax *= beta;
    }
}
```

表20.1.1给出由 **machar** 返回的一些典型值。表中 IEEE 依从机是指大多数 UNIX 工作站(SUN、DEC、MIPS),及 Apple Macintosh 上。带浮点协处理器的 IBM PC 机,一般是 IEEE 依从机,但是有些编辑器会不适当地下溢出中间结果,导致 irnd =2 而不是 5 。注意,对于 VAX(第四列)来说,它是采用在尾数中有一位"幻象"的前导位的记数法,因此对于同样字长虽然产生一个较小的 eps,但处理下溢的方式不适当。

**表20.1.1 程序 wachar 返回的典型结果**

| 精度 | 典型的 IEEE 依从机 单 | 典型的 IEEE 依从机 双 | DECVAX 单 |
|---|---|---|---|
| ibeta | 2 | 2 | 2 |
| it | 24 | 53 | 24 |
| machep | −23 | −52 | −24 |
| eps | $1.19\times10^{-7}$ | $2.22\times10^{-16}$ | $5.96\times10^{-8}$ |
| negep | −24 | −53 | −24 |
| epsneg | $5.96\times10^{-8}$ | $1.11\times10^{-16}$ | $5.96\times10^{-8}$ |
| iexp | 8 | 11 | 8 |
| minexp | −126 | −1022 | −128 |
| xmin | $1.18\times10^{-38}$ | $2.23\times10^{-308}$ | $2.94\times10^{-39}$ |
| maxexp | 128 | 1024 | 127 |
| xmax | $3.40\times10^{38}$ | $1.79\times10^{308}$ | $1.70\times10^{38}$ |
| irnd | 5 | 5 | 1 |
| ngrd | 0 | 0 | 0 |

参考文献和进一步读物:


Cody, W.J. 1988, *ACM Transactions on Mathematical Software*, vol. 14, pp. 303~311.[1]

Malcolm, M.A. 1972, *Communications of the ACM*, vol. 15, pp. 949~951. [2]

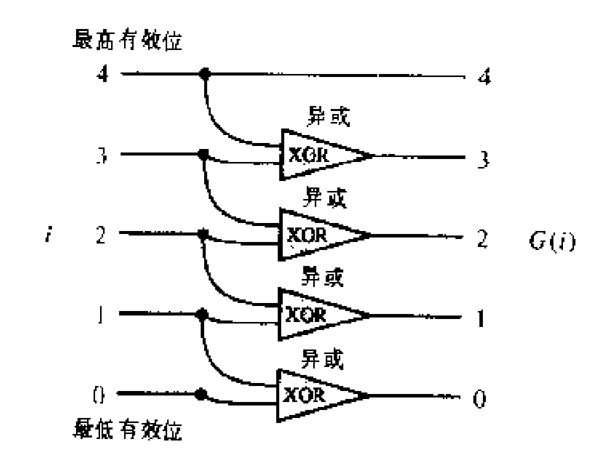
*IEEE Standard for Binary Floating-Point Numbers*, ANSI/IEEE Std 754-1985 (New York: IEEE, 1985). [3]

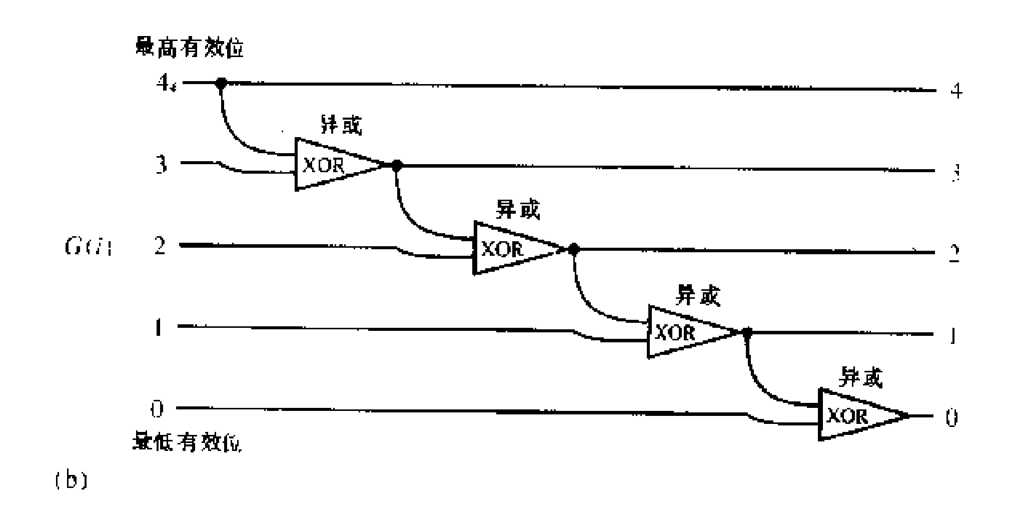

## 20.2 格雷码

格雷码是整数 $i$ 的一个函数 $G(i)$,对于每一个整数 $N\geq0$,有 $0<i\leq2^N-1$ 一一对应的 $G(i)$。该函数有以下显著特性:表达 $G(i)$ 和 $G(i+1)$ 的二进制数中只有一位二进数不同。格雷码的一个(实际上是最常用的)例子是序列0000,0001,0011,0010,0110,0111,0101,0100,1100,1101,1111,1110,1010,1011,1001和1000,分别对应于 $i=0,\cdots,15$。产生这个码

的方法是，简单地求 $i$ 与 $i/2$(整数部分)的逐位异或(XOR)(图20.2.1(a))。请想一想，若把1加到一个二进制数上时，加法是如何进行的，就会明白异或怎么进行的。从这个序列可以看出，相邻的格雷码只有一位二元数不同(如有必要，可在前缀加一个零)。

格雷的拼写是“Gray”，而不是“gray”，这种码是因 Frank Gray 而得名的，是他第一个申请关于使用轴轮或编码器想法的专利。这种编码器是一个带有许多同心编码条纹的轮子，每一个条纹的码位经固定导电刷而“读出”。这种想法是，产生一个描述轮子角度的二进制码。制造轴轮式编码器的一个显然的，但却错误的方法是，使轮子中一个(例如最内部的)条纹的一半与电刷导通，而另一半与电刷绝缘；下一个条纹在一、三象限导通；再下一个条纹在一、三、五、七的八等分区间内导通；依此类推。所有导电刷加在一起，就直接读出了轮子位置的二进制码。




(a)从 $i$ 计算格雷码 $G(i)$ 的逐位运算；(b)格雷码的译码运算。

图20.2.1

这种方法之所以差，是因为不能保证轮子转动时所有电刷能精确同步地接通或断开。当从位置7(0111)走到位置8(1000)时，由于不同导电刷不能同时导通或断开，就造成了编码虚假，会短暂地出现6(0110)、14(1110)和10(1010)。而用格雷码来编制条纹，可以确保在7

(上述序列中的0100)与 8(1100)之间没有临时状态会出现。

当然,我们需要用电路或算法把 $G(i)$ 的二进序列译回成 $i$。图20.2.1(b)显示了怎样通过异或门的级联来完成此事,想法是,每一个输出位都应是所有有效输入位的异或。对 $N$ 位格雷码进行译码,在电路中需要经过 $N-1$ 步(或者门延迟)(尽管如此,这在电路中通常是非常快的)。在一个具有一字节宽度的二进制运算的寄存器中,我们不必做 N 次连续运算,而只需做 $\ln_2 N$ 次。其中秘诀是,使用异或联合,并有层次地把运算分组。这涉及到按 1,2,4,8,…位连续右移直到字长结束,下面是一段求格雷码及其译码的程序。

```
unsigned long igray (unsigned long n, int is)
    当 is 值为正或零,返回 n 的格雷码;若 is 为负,则返回格雷码的译码 n。
{
    int ish;
    unsigned long ans, idiv;

    if (is >= 0)                              这是简单方向:
        return n ^ (n >> 1);
    ish=1;                                    这是较复杂的方向:在层次阶段中,
    ans=n;                                    从一个一位右移开始,运行每一位与所有较高有效位相异或
    for (;;) {
        ans ^= (idiv=ans >> ish);
        if (idiv <= 1 || ish == 16)           返回 ans
       ish <<= 1;                             在下一轮循环中加倍移位的量
    }
}
```

在数值计算中,当执行的任务密切地依赖于 $i$ 的二进制数,并且需对许多 $i$ 值循环执行时,格雷码是非常有用的。如果对只相差一位二进位的数,重复执行某任务能提高效率的话,那么使用格雷码顺序而不是使用一般二进数的连续顺序将会很有意义。在第7.7节中,我们已看到了用格雷码顺序产生伪随机序列的例子。

参考文献和进一步读物:


Horwitz, p., and Hill, W, 1989, *The Art of Electronics*, 2nd ed. (New York: Cambridge University Press), §8.02.

Knuth, D. E. *Combrinatorial Algorithms*, vol. 4 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §7.2.1. [Unpublished. Will it be always so?]


## 20.3 循环冗余度校验和其它的校验和式

当从 A 传送一串二元序列到 B 时,发送者希望知道它是否能准确无误地传送到达 B。一个通常的可靠方式是,在 7 位的 ASCII 字符上附加一位"奇偶校验位",使它们变成 8 位格式。这位奇偶校验位的选择是,使序列中含"1"(相对于"0")的总数或为偶("偶校验")或为奇("奇校验")。因此,就能检测出任何字符中的任何单位错误。当错误足够少,并且在时间上不是成群出现时,使用奇偶校验码就可提供足够的错误检测。

不幸的是,在实际情况中,单独噪声"事件"可能会破坏多于一位二进位。由于,奇偶校验位有两个可能值(0和1),因此平均地说,检测出多余一位二进位的错字符只有 50% 的机会。对大多数应用来说,50% 的检错概率都是不够好的。大多数通信协议中,使用对奇偶

位的一种多位推广，这种推广称作为"循环冗余度校验"或CRC。在典型的CRC应用中，有16位长（两个字节或两个字符），所以发生未被检测到随机错误的机会是 $2^{16}=65536$ 分之一。而且对任意长度的消息来说 $M$ 位CRC具有能检测出所有小于等于 $M$ 个连续位中错误的数学特性。由于通信信道中噪声倾向于"突发"，伴随相连位的短序列被破坏，因此这种连续位纠错特性是非常需要的。

通常，CRC出现在通信软件专家和芯片设计者——这些对二进数了如指掌的人——的工作领域中。但是，至少了解关于CRC的两种情况，对于我们是有用的。首先，我们有时需要能与低层硬件片或软件段通讯，这些硬件片或软件段需要有效的CRC作为其输入的一部分的。例如，用一个程序把XMODEM或Kermit软件包直接连接到通信线中，而不必将它存于局部文件中，这将是非常方便的。

第二，在大量的（例如实验）数据的处理中，能用数据集合（数字、记录、行或完整文件）的统计上独有的信息标号——它的CRC来对其进行标记，这也是有用的。任意大小的集合都可只比较其短的CRC信息标号来比较其一致性，不同的信息标号意味着不同的记录，一致的信息标号，在很高的统计确定性上，意味着一致的记录。若无法容忍小的错误概率，那么在信息标号一致时还可做完整的记录比较。当文件或数据记录有可能被偶然地或不负责任地修改（例如计算机病毒）时，把它们的先验的CRC存储于外部的、物理上安全的介质，如软盘上，是很有用的。

有时，CRC可用于记录数据时对其进行压缩。如果一致的数据记录经常出现，就可将原先遇到的记录的CRC存于内存。如果一个新记录的CRC是不同的，则就将新记录完整地归档。否则，就只有一个指针指向原先需归档的记录。在这项应用中，需要一个4或8字节的CRC，以使错误地放弃一个不记录的可能性小到可容忍的程度。或者，如果能够随机地存取原先的记录，则可以做一个完整的比较，以决定该具有一致的CRC的记录是否真的一致。

现在，让我们来简要地讨论CRC理论。然后，我们将提供各种（有关的）CRC的实施，这些CRC用于正式的或事实上的标准协议中，见表20.3.1。

CRC约定数学基础是"模2整数的多项式"。任何二进制消息可被认为是常数为0或1的多项式。例如，消息"110001101"是多项式 $x^8+x^7+x^3+x^2+1$。由于0和1是模2的仅有的整数，所以多项式中x的幂次项或者存在(1)，或者不存在(0)。这些模2整数的多项式称为唯一因式分解域。它的含义是，任何多项式有一个唯一的因式分解，即被称为不可约的或"本原"多项式——类似于质数。多项式 $x^2+x+1$ 是本原多项式，而多项式 $x^2+1$ 则不是本原的：$x^2+1=(x+1)(x+1)$。请记住，整数运算需以2为模运算！一个有关的定理说，如果一个多项式 $p$ 除尽另一个多项式 $q$，则它一定能除尽至少 $q$ 中的一个因子。

一个 $M$ 位长的CRC是基于一个特殊的 $M$ 次本原多项式，称它为生成多项式。选用哪一个本原多项式作为生成多项式只是一种习惯上的事。对于16位CRC，CCITT（Comité Consultatif International Télégraphique et Téléphonique）已规定"CCITT多项式"为 $x^{16}+x^{12}+x^5+1$。列在表20.3.1中的所有协议都使用这个多项式。另一个通常选用者"CRC-16"多项式，$x^{16}+x^{15}+x^2+1$，它被用于IBM的BISYNCH中的EBCDIC消息中。一种通常的12位选择，"CRC-12"，是 $x^{12}+x^{11}+x^3+x^2+x+1$。一种通常的32位选择，"AUTODIN-II"是 $x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$。关于一些其它本原多项式的表格，参见第7.4节。

表20.3.1　各种 CRC 协议的约定和测试值

| | crc 参数 | | 测试值($C_2C_1$ 为16进制数) | | 包 | |
|---|---|---|---|---|---|---|
| 协议 | jinit | jrev | T | CatMouse987654321 | 格式 | CRC |
| XMODEM | 0 | 1 | 1A71 | E556 | $S_1S_2\ldots S_NC_2C_1$ | 0 |
| X.25 | 255 | −1 | 1B26 | F56E | $S_1S_2\ldots S_N\overline{C_1}\overline{C_2}$ | F0B8 |
| (无名) | 255 | −1 | 1B26 | F56E | $S_1S_2\ldots S_NC_1C_2$ | 0 |
| SDLC(IBM) | 与 X.25同 | | | | | |
| HDLC(ISO) | 与 X.25同 | | | | | |
| CRC—CCITT | 0 | 1 | 14A1 | C28D | $S_1S_2\ldots S_NC_1C_2$ | 0 |
| 无名 | 0 | −1 | 14A1 | C28D | $S_1S_2\ldots S_N\overline{C_1}\overline{C_2}$ | F0B8 |
| Kermit | 与 CRC—CCITT 同 | | | | 见注释 | |

注释：上划线表示补码。$S_1\ldots S_N$ 是字符数据。$C_1$ 是 CRC8的位最低有效位，$C_2$ 是 8 位最高有效位，因此，$CRC=256C_2+C_1$(用16进制表示)。Kermit(块校验水平 3)把 CRC 作为 3 个可打印的 ASCII 字符传送(传送值−32)。这些分别包含 4 位最高有效位、6 位中间位、6位最低有效位。

已知 $M$ 次生成多项式 $G$(可被写成多项式形式或二进位序列，例如，CCITT 多项式可写成10001000000100001)，计算一个二进制序列 $S$ 的 CRC 的方法如下：首先，用 $x^M$ 乘 $S$，也就是把 $M$ 个零位添到 $S$ 序列后面。第二，用 $G$ 长除 $Sx^M$。记住长除中的减法是模 2 运算，因此永远不会有"借位"，模 2 减法与逻辑上的异或($XOR$)相同。第三，忽略所得的商数。第四，最终得到一个余数，这就是循环冗余校验式称之为 $C$。$C$ 将是一个小于等于 $M$ 次的多项式，否则就还没有完成长除。因此，在二进位序列形式中，循环冗余校验式 $C$ 的二进序列有 $M$ 位，可能包括前导位为零($C$ 甚至可能全为零，见下文)，实例见[3]。

如果在一个例子中按上述步骤进行，读者会看到，在长除表中写下的东西是多余的。实际上，只需把 $S$ 的连续位从右向左移至一个 $M$ 位的寄存器中。每一次有一位移出寄存器的左端，通过将它与 $G$ 的 $M$ 个低位(也就是说，除头一位外的所有 $G$ 的位)进行异或，并改变这个寄存器。当一个零位移出寄存器左端时，寄存器将不改变。当 $S$ 的原序列部分的最后一位移出寄存器左端后，寄存器中所剩下的就是 $S$ 的 CRC。

由此看到，这些过程用硬件实现是很有效的，它只要求一个连接一些异或门的移位寄存器。这就是在通信设施中，如何由单片芯片(或一片芯片的一小部分)来计算 CRC 的。在软件中，实现起来就没有这么容易了，因为二进位的移动一般不是很有效的。因此，人们发明(正如下面的实施)了表格驱动程序，它可预先计算一系列移位和异或的结果。例如，8 位输入中所有 256 种可能的序列的每一种。

现在，我们讨论 CRC 如何获得对 $M$ 个连续位中所有错误的检测能力。假设有两则消息，$S$ 和 $T$，它们只是两个 $M$ 位框架内不同的序列。于是，它们的 CRC 只相差一个数，即 $G$ 除 $(S-T)x^M\equiv D$ 的余数。现在，$D$ 的形式是在 $M$ 位框架序列中，前导位为零(可忽略)，紧跟一些1，后跟尾随的零(正好是 x 的乘法因子)。由于因式分解是唯一的，所以 $G$ 不可能除尽 $D$：$G$ 是 $M$ 次的本原多项式，而 $D$ 是 $x$ 的幂次乘上一个(至多)$M-1$次因式。因此，$S$ 和 $T$ 不可避免地具有不同的 CRC。

在许多协议中，一个传递的数据块由 $N$ 数据位和后面直接跟着它们的 $M$ 位循环冗余校验位(或与一个常数异或的 CRC)组成。在接收端，恢复数据块有两种等价方法。最明显的方法是，在接收端计算数据位的 CRC，并将其与传递的 CRC 相比较。另一种不太明显的、但更精巧的方法是，接收端简单地计算整个 $N+M$ 位数据块的 CRC，并确证得到了零结果。证明：完整的数据块是多项式 $Sx^M+C$(数据左移，以便留出 $M$ 位 CRC 的位置)。而 $C$ 的定义是 $Sx^M=QG+C$，其中 $Q$ 是放弃的商数。所以，$Sx^M-C=QG-C-C=QG$(请记住，模 2 运算)，它是 $G$ 的完整的倍数。因此，当在接收端，将整个数据块乘以 $x^M$ 时，它仍由 $G$ 的乘积，因此它的 CRC 为零。证毕。

这个基本程序中有一两处小变化需要提一下[5]。首先，在计算 CRC 时，$M$ 位寄存器不需初始化为零。将它初始化为其它 $M$ 位值(例如全为1)，其在效果上等于所有块的前端加了一个虚假消息，即把初始化值作为余值给出。这样做是有优点的，因为不如此描述 CRC 就不能检测出对起始零位个数的增减。(丢失起始位、或插入零位是普通的"时钟错误")。第二，可以在 CRC 被传递之前加上(异或)任何 $M$ 位常数 $K$。这个常数或在接收端被异或掉，或只是对全部块的 CRC 改变一个已知量，即 $G$ 除 $Kx^M$ 余数。常数 $K$ 经常有"所有位"，它把 CRC 变换成其补码。这样做有一个优点就是，能检测出 CRC 不能发现的另一种错误：能检测出在块结尾处突然插入一位"1"，而使消息的起始位为"1"的错误。

下面函数 **icrc** 实现了上述 CRC 计算，并且包括了所提及的变化。该函数的输入是指向一个字符数组的指针以及该数组的长度。

**icrc** 有两个"开关"参数，它详细规定了 CRC 计算中的变化。**jinit** 取零值或正值，会引起16 位寄存器用 jinit 值，对它的每一个字节进行初始化。jrev 取负值，会引起每一个输入字符被译成它的比特位倒置像，并且在输出的 CRC 中，也完成相似的比特位倒置。读者不必非弄懂这些内容不可，仅仅使用表20.3.1规定的 jinit 和 jrev 值就行了。(其简单解释是，串行数据端口是先传送字符的最低有效位，并且许多协议准确地按接收顺序将二进数移进 CRC 寄存器。)表20.3.1显示了如何由 **icrc** 的输入数组和输出 CRC 来构造一个字符块。无需在 **icrc** 之外做任何附加的比特位倒置。

开关 jinit 有一个附加用途：当出现负值时，用数组 crc 的输入值初始化寄存器。如果设置 crc 为最后一次调用 **icrc** 的结果，这实际上是把当前输入数组连接到前次调用中的数组上。使用这个特性，可一次一行地为整个文件建起 CRC，而不必把整个文件都保存在内存中。

程序 **icrc** 基本是根据[4]中函数编写的。其操作可理解如下：首先看看函数 **icrc1**。它把一个输入字符输入一个 16 位 CRC 寄存器。这里用到的唯一的技巧是，在寄存器每次移位时，把字符位异或成最高有效位，一次八位；而不是一次一位地送入最低有效位，并一次异或一位。这是因为异或是关联的和可交换的——我们可以在决定是否用本原生成多项式改变寄存器之前的任何时候，输进字符位。(十进制常数4129是对应于生成多项式除去最高位的二进制数。)

unsigned short icrc1(unsigned short crc, unsigned char onech)

给定现在的余数，返回一个字符加入后的新 CRC，其功能等价于 **icrc**(,,1,—1,1)，但速度较慢，一般它为 **icrc** 初始化表格所用。

```
    int i;
    unsigned short ans=(crc ^ onech << 8);

    for (i=0;i<8;i++) {                    在此完成8个一位位移,并与生成多项式异或
        if(ans & 0x8000)
            ans = (ans <<= 1) ^ 4129;
        else
            ans <<=1;
    }
    return ans;
}
```

现在看一看 icrc。有两部分内容要弄懂,当初始化时如何建立表,此后如何使用该表。回想一下,把字符的位从最低有效位末端移入 CRC 寄存器。关键的监测是,当 8 位二进数移进寄存器低端时,所有生成的要改变的位都已经由已存于高端的位所决定。由于异或是可交换的和关联的,所以我们所需的是,对每一个 256 种可能的高位序列,进行由生成多项式作改变的所有结果表。然后,我们可以进行捕捉和把输入字符异或成这个数中查找的结果。

icrc 的另一内容是,在初始化时从存于 it 中的 4 位表来构造一个 8 位比特位倒置的表结构,以及与执行比特倒置有关的逻辑。参考文献[4~6]给出表格驱动的 CRC 计算的细节。

```
typedef unsigned char uchar;
#define LOBYTE(x) ((uchar)((x) & 0xFF))
#define HIBYTE(x) ((uchar)((x) >> 8))

unsigned short icrc(unsigned short crc, unsigned char *bufptr,
    unsigned long len, short jinit, int jrev)
    对 len 字节长的数组 bufptr 计算其16位循环冗余校验,它是通过采用按某一种约定来设置 jinit 和 jrev(见表20.3.
    1)进行计算的。若 jinit 取负值,则 crc 作为输入以便初始化余数寄存器;事实上,(crc 设置为上次调用的返回值)将
    输入 bufptr 数组连接到上一次调用上。
{
    unsigned short icrc1(unsigned short crc, unsigned char onech);
    static unsigned short icrctb[256],init=0;
    static uchar rchr[256];
    unsigned short j,cword=crc;
    static uchar it[16]={0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15};     4位比特位倒置后的数值表

    if (!init) {                                                     需要初始化表格
        init=1;
        for (j=0;j<=255;j++) {                                       这两个表是:所有字符的CRC,
            icrctb[j]=icrc1(j << 8, (uchar)0);                       和所有字符的比特位倒置
            rchr[j]=(uchar)(it[j & 0xF] <<4 | it[j >> 4]);
        }
    }
    if (jinit >= 0) cword=((uchar) jinit) | (((uchar) jinit) << 8);  初始化余数寄存器
    else if (jrev < 0) cword=rchr[HIBYTE(cword)] | rchr[LOBYTE(cword)] << 8;    如果不初始
                                                                                化,需倒置
                                                                                寄存器吗?
    for (j=1;j<=len;j++)                                             对数组中字符的主循环
        cword=icrctb[(jrev < 0 ? rchr[bufptr[j]] :
            bufptr[j]) ^ HIBYTE(cword)] ^ LOBYTE(cword) << 8;
    return (jrev >= 0 ? cword : rchr[HIBYTE(cword)] | rchr[LOBYTE(cword)] << 8);   需对输出
                                                                                倒置吗?
}
```

若需要 32 位校验和式,怎么做呢?对一个真正的 32 位 CRC,就需要用一个更长的生

成多项式来重写此程序。例如，$x^{12}+x^7+x^5+x^3+x^2+x+1$ 是模 2 本原(多项式)，并且(为了简单)仅在它的最低有效的字节中具有非零的前导位。因此只对 CRC 寄存器的最高有效位的字节进行表查找的思想没有改变。

如果不关心校验 $M$ 个连续位的特性，而只是需要校检一个统计随机的 32 位，那么，可使用此处给出的 **icrc**；以 jrev=1 调用它一次以得到 16 位，再以 jrev=－1 得到另一个 16 位。内部的比特位倒置使得这两个 16 个 CRC 彼此完全独立。

### 20.3.1 其它种类的校验和式

与 CRC 非常不同的技术是，常常在人们处理的(例如键入计算机的)数字上添加一个十进制的“校验位”，校验位需要用来抵挡人们倾向于犯有高度结构性的错误，例如传送连续数字。瓦格纳(Wagner)和普特(Putter)[7]就这个问题给出了有意义的介绍，包括详细的算法。

目前，所有广泛传播的校验和式，从中等到低劣，非常相同。在大多数书上，包括本书上的 10 位 ISBN(国际标准书号)，都使用的校验方程为

$$10d_1+9d_2+8d_3+\cdots+2d_9+d_{10}=0 \quad (\text{模 } 11) \qquad (20.3.1)$$

其中 $d_{10}$ 是右端检验位。字符“X”用于代表校验位的数值 10。另一种流行的格式是所谓的“IBM 校验”，常用于记数(包括，例如硕士学位证件)。这里，校验方程是

$$2\#d_1+d_2+2\#d_3+d_4+\cdots=0 \quad (\text{模 } 10) \qquad (20.3.2)$$

其中 2# 的含义是，“用 2 乘 $d$ 并加上结果的十进制数字”。美国银行码用一个 9 位数进行校验，它的校验方程是

$$3a_1+7a_2+a_3+3a_4+7a_5+a_6+3a_7+7a_8+a_9=0 \quad (\text{模 } 10) \qquad (20.3.3)$$

印在许多美国邮政服务信封上的条形码是通过去掉两端的等高标记的条形，然后 5 条一组，把剩余的条形成 6 组或 10 组而进行解码的。在每一组中，五条条形(从左到右)分别表示数值 7，4，2，1，0。它们当中准确地有两个是高的，它们之和就是代表数字，除了零是由 7+4 来代表的。5 位或 9 位的邮政编码后跟一个校验位，校验方程是

$$\sum d_i=0 \quad (\text{模 } 10) \qquad (20.3.4)$$

这些格式中，没有一个是接近最优的。Vorhodff 提出一个精致的格式，它叙述在文献[7]中。其基本思想是使用十个元素的**二面体群** $D_5$ ——它对应于五边形的对称性——而不是使用模 10 整数的循环群。这个校验的方程是

$$a_1 * f(a_2) * f^2(a_3) * \cdots * f^{n-1}(a_n)=0 \qquad (20.3.5)$$

其中 * 是 $D_5$ 中的(不可交换的)乘法，且 $f^i$ 表示对某固定排列的第 $i$ 次迭代。Verhoeff 的方法可发现一串字符中的所有单个错误和所有相连换位错误。它还可发现大约 95% 的成对错误($aa\to bb$)、跳换的错误($acb\to bca$)和成对跳换的错误($aca\to bcb$)。下面是一种实施方案。

```
int decchk (char string[], int n, char *ch)
    计算和验证十进制校验数字。返回一个校验数字 ch，将它附加到 string[1..n]上，即存入 string[n+1]中。在本模式中，忽略所返回的布尔(整数)值。若 string[1..n]已经用一校验位作结尾，则返回函数值为真(1)；若没有校验位，则为假(0)。在本模式中忽略 ch 的返回值。注意，string 和 ch 是数字0～9对应的 ASCII 字符，不是0～9范围中的字节值。string 中也允许有其它 ASCII 字符，但在计算校验数字时它是被忽略的。
{
    char c;
    int j,k=0,m=0;
```

```
    static int ip[10][8]={0,1,5,8,9,4,2,7,1,5, 8,9,4,2,7,0,2,7,0,1,
        5,8,9,4,3,6,3,6, 3,6,3,6,4,2,7,0,1,5,8,9, 5,8,9,4,2,7,0,1,6,3,
        6,3,6,3,6,3,7,0,1,5, 8,9,4,2,8,9,4,2,7,0, 1,5,9,4,2,7,0,1,5,8};
    static int ij[10][10]={0,1,2,3,4,5,6,7,8,9, 1,2,3,4,0,6,7,8,9,5,
        2,3,4,0,1,7,8,9,5,6, 3,4,0,1,2,8,9,5,6,7, 4,0,1,2,3,9,5,6,7,8,
        5,9,8,7,6,0,4,3,2,1, 6,5,9,8,7,1,0,4,3,2, 7,6,5,9,8,2,1,0,4,3,
        8,7,6,5,9,3,2,1,0,4, 9,8,7,6,5,4,3,2,1,0};          群乘积和排列表

    for (j=0;j<n;j++) {                                       查看连续字符
        c=string[j];
        if (c >= 48 && c <= 57)                               除数字外忽略其它
            k=ij[k][ip[(c+2) % 10][7 & m++]];
    }
    for (j=0;j<=9;j++)                                        找出附加的一致校验数
        if (!ij[k][ip[j][m & 7]]) break;
    *ch=j+48;                                                 转换成 ASCII 码
    return k==0;
}
```

参考文献和进一步读物：


McNmara, J. E. 1982, *Technical Aspects of Data Communication*, 2nd ed. (Bedford, MA: Digital Press). [1]

da Cruz, F. 1987, *Kermit, A File Transfer Protocol* (Bedford, MA: Digital Press). [2]

Morse, G. 1986, *Byte*, vol. 11, pp. 115~124 (September). [3]

LeVan, J, 1987, *Byte*, vol. 12, pp. 339~341 (November). [4]

Sarwate, D. V. 1988, *Communications of the ACM*, vol. 31, pp. 1008~1013. [5]

Griffiths, G. and Stones, G. C. 1987, *Communications of the ACM*, vol. 30, pp617~620. [6]

Wagner, N. R., and Putter, P. S. 1989, *Communications of the ACM*, vol. 32, pp. 106~110. [7]


## 20.4 霍夫曼码与数据压缩

一个不丢失数据的压缩算法是将一串符号(一般为 ASCII 字符或字节)可逆地转换成另一串符号，而在长度上平均地要短一些。"平均地"这含意是很重要的，很显然没有可逆的算法能把所有的字符串都变短——因为没有足够多的短串与较长的串一一对应。只有当在输入端，有一些串或输入符号，比起其它的符号要更常用时，压缩算法才是可能的。于是这些常见的串用短的序列，而比较少见的串或符号用较长的串进行编码，这样在平均意义上长度为最短。

对应于不同的检测方法和在输入串中使用不同的概率偏离的方法，有许多压缩编码的技术，在这节和下一节中，我们只考虑以**定义字**输入的**变长码**。在这些码中，输入被分成固定长度单位，例如 ASCII 字符，而相应输出为可变长度块。这种最简单的方法就是本节中讨论的霍夫曼(Huffman)编码[1]。另一个例子，**算术压缩**编码，将在第20.5节中讨论。

与定义字变长码相反的方法是，把输入分成变长的单位(例如，英语课文中的单字或词组)，然后经常以固定长度的输出码将它们传输。这种类型中，最常用的编码是 Ziv-Lempel 编码[2]。参考文献[3～6]给出了某些压缩技术，并列出有关的大量参考文献。

霍夫曼编码的思想很简单，它是对较常用的字符使用较短的位图样式。根据**熵**的概念，可使这种思想定量化。假设输入字母表中有 $N_{ch}$ 个字符，分别以概率 $p_i(i=1,\cdots,N_{ch})$ 出现在输入字符串中，因此 $\sum p_i = 1$。根据信息论的基本定理，由这些字符组成的相互独立的

随机序列（一种保守，但并不总是现实的假设）平均地要求每个字符至少是 $H$ 位：

$$H = -\sum p_i \log_2 p_i \qquad (20.4.1)$$

其中 $H$ 是概率分布的熵。另外，一定存在这种可任意接近这个边界值的编码方式。对于所有字符 $p_i=1/N_{ch}$ 等概率的情况，很容易得出 $H=\log_2 N_{ch}$，它是无压缩的情况。$p_i$ 的任意其它集合给出较小的熵，它允许进行有效的压缩。

注意，如果我们用长度为 $L_i=-\log_2 p_i$ 位的码对字符 i 进行编码，这样方程式(20.4.1)将平均为 $\sum p_i L_i$，式(20.4.1)的边界就可以达到。这种编码方式的麻烦是，一般 $-\log_2 p_i$ 不是一个整数。我们怎样能把字母"Q"编码成 5.32 位长的码字呢？霍夫曼编码作了一种尝试，即实际上用 1/2 的整数次幂近似所有的概率 $p_i$，因此所有 $L_i$ 都为整数。如果所有 $p_i$ 实际上都是 2 的负整数次幂，则霍夫曼码的确达到熵边界 $H$。

霍夫曼码的构造最好用举例来说明。想象一种语言，Vowellish，它有 $N_{ch}=5$ 种字符的字母表 A、E、I、O 和 U，分别以 0.12、0.42、0.09、0.30 和 0.07 的概率出现。于是为 Vowellish 构造的霍夫曼码按表20.4.1中的步骤来完成。

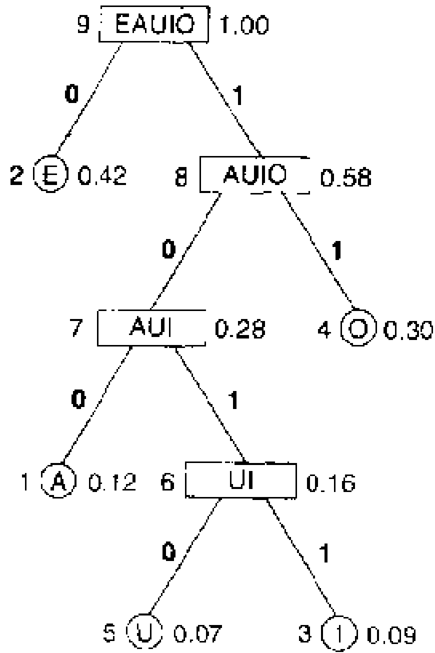
表20.4.1 Huffman码的构造步骤表

| 节点 | 阶段： | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 1 | A: | 0.12 | 0.12■ | | | |
| 2 | E: | 0.42 | 0.42 | 0.42 | 0.42■ | |
| 3 | I: | 0.09■ | | | | |
| 4 | O: | 0.30 | 0.30 | 0.30■ | | |
| 5 | U: | 0.07■ | | | | |
| 6 | | UI: | 0.16■ | | | |
| 7 | | | AUI: | 0.28■ | | |
| 8 | | | | AUIO: | 0.58■ | |
| 9 | | | | | EAUIO: | 1.00 |

它是通过 $N_{ch}$ 个阶段顺序进行的，由表中的列表示。第一阶段从 $N_{ch}$ 个节点开始，每一节点表示一个字母表中的字母，分别包含它们的相应概率。在每一阶段，将两个最小的概率，加合起来构成一个新的节点，然后将这新节点写在新的一列下面。（表中小黑方块表示该节点已合并，并在下阶段中丢弃）。所有活动节点（包括新合成的）都列入下一阶段中。在表20.4.1中，赋给新节点名称（比如 AUI）是不连贯的。在所举例子中（阶段 1 之后），碰巧每阶段的两个最小节点总是一个是原始节点，另一个是合成的节点。一般情况，两个最小概率可能都是原始节点，或都是合成的，或各占一个。在最后阶段，所有节点都要被集中到一个概率为 1 的总合成中。

现在，为了得到这些码字，把表20.4.1 中的数据重新画成一棵树（图20.4.1），如图所

示，树中的每一节点对应于表中的一个节点(行)，其左边标着节点的整数序号，右边标着它的概率值。所谓终端节点由圆圈表示，它们是单个字母表的字符。在树枝上标记 0 和 1。字符的码字就是从上到下指向它的 0 和 1 的序列。例如 E 的码字简单地为 0，而 U 的码字是 1010 。




沿着树从顶向下移动，对字母(A、E、I、O 和 U)进行译码的编码。码字是所经过树枝上所对应的0或1的序列。每个端点右边的值是它的概率，左边的值是表20.4.1中的端点编号。

图20.4.1 虚构语言 Vowellish 的霍夫曼码，以符号形式表示。

现在，任意一串 0、1 符号串都能译成对应的字母序列。例如考虑符号串 1011111010 ，从树的顶端开始下降经过 1011 到达 I ——第一个字符。由于我们已到达了一个终端节点，所以重置于树的顶端，下一次经过 11 下降到 O。最后，1010 给出 U。因此符号串译成为 IOU 字母串。

这些思想体现在下面的程序中。第一个程序 **hufmak** 的输入是，字母表的 nchin$=N_{ch}$个字符发生频率的整数向量，即与 $p_i$ 成正比的整数集。**hufmak** 和它所调用的 **hufapp**，实行了表20.4.1的构造和图20.4.1中树的构造。为了有效性，这个程序使用堆结构(见第8.3节)。关于详细描述，参见文献[7]。

```
#include "nrutil.h"

typedef struct {
     unsigned long *icod, *ncod, *left, *right, nch, nodemax;
} huffcode;

void hufmak(unsigned long nfreq[], unsinged long nchin, unsigned long *ilong,
     unsigned long *nlong, huffcode *hcode)
     给定 nchin 个字符的发生频率表 nfreq[1..nchin]，在结构 hcode 中构造霍夫曼码。返回值 ilong 和 nlong，它们是所产
```

生最长的符号的字符编码及其码长，并应该检查 nlong 不大于机器的字长。

```
{
    void hufapp(unsigned long index[], unsigned long nprob[], unsigned long n,
        unsigned long i);
    int ibit;
    long node,*up;
    unsigned long j,k,*index,n,nused,*nprob;
    static unsigned long setbit[32]={0x1L,0x2L,0x4L,0x8L,0x10L,0x20L,
        0x40L,0x80L,0x100L,0x200L,0x400L,0x800L,0x1000L,0x2000L,
        0x4000L,0x8000L,0x10000L,0x20000L,0x40000L,0x80000L,0x100000L,
        0x200000L,0x400000L,0x800000L,0x1000000L,0x2000000L,0x4000000L,
        0x8000000L,0x10000000L,0x20000000L,0x40000000L,0x80000000L};

    hcode->nch=nchin;                                    初始化
    index=lvector(1,(long)(2*hcode->nch-1));
    up=(long *)lvector(1,(long)(2*hcode->nch-1));      将保持堆的径迹的向量
    nprob=lvector(1,(long)(2*hcode->nch-1));
    for (nused=0,j=1;j<=hcode->nch;j++) {
        nprob[j]=nfreq[j];
        hcode->icod[j]=hcode->ncod[j]=0;
        if (nfreq[j]) index[++nused]=j;
    }
    for (j=nused;j>=1;j--) hufapp(index,nprob,nused,j);
    Sort nprob into a heap structure in index.
    k=hcode->nch;
    while (nused > 1) {                                  连接堆的节点，在每一步中重新
        node=index[1];                                       修改堆
        index[1]=index[nused--];
        hufapp(index,nprob,nused,1);
        nprob[++k]=nprob[index[1]]+nprob[node];
        hcode->left[k]=node;                             存储一个节点的左和右的子码
        hcode->right[k]=index[1];
        up[index[1]] = -k;                               指示一个节点是否是左和右子码
        up[node]=index[1]=k;                                 的父码
        hufapp(index,nprob,nused,1);
    }
    up[hcode->nodemax=k]=0;
    for (j=1;j<=hcode->nch;j++) {                        从树构造霍夫曼码
        if (nprob[j]) {
            for (n=0,ibit=0,node=up[j];node;node=up[node],ibit++) {
                if (node < 0) {
                    n |= setbit[ibit];
                    node = -node;
                }
            }
            hcode->icod[j]=n;
            hcode->ncod[j]=ibit;
        }
    }
    *nlong=0;
    for (j=1;j<=hcode->nch;j++) {
        if (hcode->ncod[j] > *nlong) {
            *nlong=hcode->ncod[j];
            *ilong=j-1;
        }
    }
    free_lvector(nprob,1,(long)(2*hcode->nch-1));
    free_lvector((unsigned long *)up,1,(long)(2*hcode->nch-1));
    free_lvector(index,1,(long)(2*hcode->nch-1));
}
```

void hufapp(unsigned long index[], unsigned long nprob[], unsigned long n, unsigned long i)

用于 **hufmak**，以在数组 index[1..]中维持堆的结构。

```
	unsigned long j,k;

	k=index[i];
	while (i <= (n>>1)) {
		if ((j = i << 1) < n && nprob[index[j]] > nprob[index[j+1]]) j++;
		if (nprob[k] <= nprob[index[j]]) break;
		index[i]=index[j];
		i=j;
	}
	index[i]=k;
}
```

注意，必须在用户的主程序中用类似以下说明来定义和分配结构 hcode：

```
#include "nrutil.h"
#define MC 512                                  hufmak 中 nchin 的最大预期值
#define MQ (2*MC-1)
typeddf struct {
	unsigned long *icod, *ncod, *left, *right, nch, nodemax;
} huffcode;
	...
	huffcode hcode;
	...
	hcode.icod=(unsigned long *)lvector(1,MQ);        为 hcode 分配空间
	hcode.ncod=(unsigned long *)lvector(1,MQ);
	hcode.left=(unsigned long *)lvector(1,MQ);
	hcode.right=(unsigned long *)lvector(1,MQ);
	for (j=1;j<=MQ;j++) hcode.icod[j]=hcode.ncod[j]=0;
```

一旦码构造完毕，可通过反复调用 **hufenc** 对字符串进行编码，程序 **hufenc** 是简单地形成码的查找表以及在输出消息中将码加入。

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
	unsigned long *icod,*ncod,*left,*right,nch,nodemax;
} huffcode;

void hufenc(unsigned long ich, unsigned char **codep, unsigned long *lcode,
	unsigned long *nb, huffcode *hcode)
	利用结构 hcode 中的码，对单个字符 ich（在0..nch-1范围内）进行霍夫曼编码，将结果写入字符数组 *codep[1..
	lcode]中，并以 nb 位起始（它的最小有效数值为零），并且适当地增加 nb。对消息中连续字符进行编码需反复调用本
	程序，但必须提前调用构造 hcode 的程序 hufmak 进行初始化。
{
	void nrerror(char error_text[]);
	int l,n;
	unsigned long k,nc;
	static unsigned long setbit[32]={0x1L,0x2L,0x4L,0x8L,0x10L,0x20L,
		0x40L,0x80L,0x100L,0x200L,0x400L,0x800L,0x1000L,0x2000L,
		0x4000L,0x8000L,0x10000L,0x20000L,0x40000L,0x80000L,0x100000L,
		0x200000L,0x400000L,0x800000L,0x1000000L,0x2000000L,0x4000000L,
		0x8000000L,0x10000000L,0x20000000L,0x40000000L,0x80000000L};

	k=ich+1;    将字符范围 0..nch-1转换成数组索引的范围 1..nch
```

```
    if (k > hcode->nch || k < 1) nrerror("ich out of range in hufenc.");
    for (n=hcode->ncod[k]-1;n>=0;n--,++(*nb)) {      在已存储的霍夫曼码中对ich进行
        nc=(*nb >> 3);                                    循环
        if (++nc >= *lcode) {
            fprintf(stderr,"Reached the end of the 'code' array.\n");
            fprintf(stderr,"Attempting to expand its size.\n");
            *lcode *= 1.5;
            if ((*codep=(unsigned char *)realloc(*codep
                (unsigned)(*lcode*sizeof(unsigned char)))) == NULL) {
                nrerror("Size expansion failed.");
            }
        }
        l=(*nb) & 7;
        if (!l) (*codep)[nc]=0;                           设置码中适当的二进位
        if (hcode->icod[k] & setbit[n]) (*codep)[nc] |= setbit[l];
    }
}
```

霍夫曼码字的译码较复杂些。必须根据变化的二进位数从码树的顶部向下[illegible]移动。

```
typedef struct {
    unsigned long *icod, *ncod, *left, *right, nch, nodemax;
} huffcode;

void hufdec(unsigned long *ich, unsigned char *code, unsigned long lcode,
    unsigned long *nb, huffcode *hcode)
```

在字符数组code[1..lcode]中，以二进位数nb起始，使用存储在结构hcode中的霍夫曼码，将码字译成单个字符(返回到处于范围0..nch-1中的ich)，并且适当地增加nb。以nb=0起始，反复调用本程序，将返回连续的字符。返回值ich=nch，表示消息的结尾。在用户的主程序中，必须已经定义和分配结构的hcode，并且也需调用**hufmak**来[illegible]

```
{
    long nc,node;
    static unsigned char setbit[8]={0x1,0x2,0x4,0x8,0x10,0x20,0x40,0x80};

    node=hcode->nodemax;
    for (;;) {                                  设置node为译码树的顶端，并循环直至得到[illegible]
        nc=(*nb >> 3);
        if (++nc > lcode) {                     用完输入，并用ich=nch指出消息的结尾
            *ich=hcode->nch;
            return;
        }
        node=(code[nc] & setbit[7 & (*nb)++] ?
            hcode->right[node] : hcode->left[node]);      从树左还是树右分叉，取[illegible]值
        if (node <= hcode->nch) {                        若到达端点则得一个完[illegible]
            *ich=node-1;
            return;
        }
    }
}
```

为了简便，当全部用完码字字节时**hufdec**就中止退出。若已编码的消息不是整数个字节，且如果$N_{ch}$小于256，则**hufdec**就能从最后码字字节的假造的结尾中，返回译码得来的最后一个或二个假造的字符。若用户掌握了有关发送字符数的其它的知识，就可轻松地放弃这些假字符。否则，可以通过提供一些二进位数，而不是一个字节数来固定这种操作，并且应相应地修改程序。(当$N_{ch}$大于等于256时，常规地，**hufdec**将在一个假造字符的中途中[illegible]

退出，并且将其它放弃掉。

### 20.4.1 游程编码

对于高度相关的二元数据流的压缩（例如沿一条传真扫描线的黑白值），经常把霍夫曼压缩与**游程编码**相结合：不再分别传递每一个比特位，而是将输入流转换成一系列整数，它用以表示有多少个相同值的连续位。然后这些整数再按霍夫曼压缩编码。CCITT 第三组根据这种方式，对一套八种标准文件规定了固定的、不可改变的、最佳霍夫曼码的专有标准[8]。

参考文献和进一步读物：


Hamming, R. W. 1980, *Coding and Information Theory*(Englewood Cliffs, NJ:Prentice-Hall).

Huffman, D. A. 1952, *Proceedings of the Institute of Radio Engineers*, vol. 40, pp. 1098~1101. [1]

Ziv, J., and Lempel, A. 1978, *IEEE Transactions on Information Theory*, vol. IT-24, pp. 530~536. [2]

Cleary, J. G., and Witten I. H. 1984, *IEEE Transactions on Communications*, vol. COM-32pp. 396~402. [3]

Welch, T. A. 1984, *Computer*, vol. 17, no. 6, pp. 8~19. [4]

Bentley, J. L., Sleator, D. D., Tarjan, R. E., and Wei, V. K. 1986, *Communications of the ACM*, vol. 29, pp. 320~330. [5]

Jones, D. W. 1988, *Communications of the ACM*, vol. 31, pp. 996~1007. [6]

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA:Addison-Wesley), Chapter 22. [7]

Hunter, R., and Robinson, A. H. 1980, *Proceedings of the IEEE*, vol. 68, pp. 854~867. [8]

Marking, M. P. 1990, *The C Users' Journal*, vol. 8, no. 6, pp. 45~54. [9]

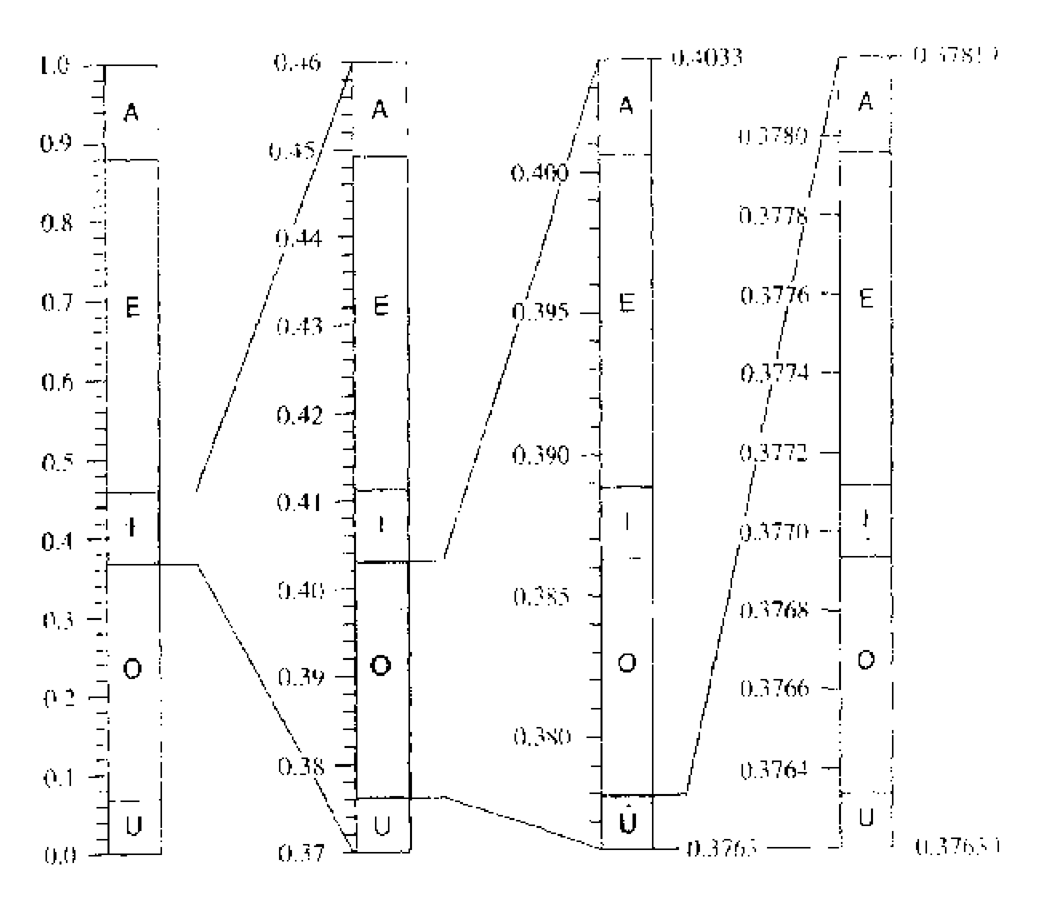

## 20.5 算术编码

在前一节中，我们看到一种完备的（以熵为界的）编码方案。若 $p_i$ 为字符 $i$（在范围 $1\leq i\leq N_{ch}$ 中）的出现概率，则用 $L_i=-\log_2 p_i$ 位来对其编码。霍夫曼编码给出了一种把 $L_i$ 舍入到接近的整数，并用这一长度来构造码的方法。现在，我们讨论**算术编码**[1]，实际上它确实试图用非整数个二进位来对字符编码。它还能提供一种方便的方法，不是把结果输出为二进位的数据流，而是输出以任何所期望的基数的符号流。这种特性在需要时特别有用，例如，把数据从字节（基数为 256 的）转换成可打印的 ASCII 字符（基数为 94）；或者转换成只包括 A～Z 和 0～9（基数为 36）的字母数字序列。

在算术编码中，把一个任意长度的输入消息表示为在 $0\leq R\leq 1$ 范围中的实数 R。消息越长，对 $R$ 的精度要求越高。通过例子能得到最好的说明，所以我们回到上一节虚构的语言 Vowellish 上。回想一下，Vowellish 有一张个 5 字符的字母表（A、E、I、O、U），出现概率分别为 0.12、0.42、0.09、0.30和0.07。图20.5.1显示了从“IOU”开始的消息是如何编码的：区间(0,1)被分成对应于 5 个字母的段，每一段的长度正比于 $p_i$。我们看到第一个消息字符，“I”，其 R 的范围为 $0.37\leq R\leq 0.46$ 。再把这个区间分成五个子区间，长度也是正比于 $p_i$。对于第二个消息字符，“O”，再细分把 R 范围变窄到 $0.3763\leq R\leq 0.4033$ 。再进一步把“U”字

符范围变窄到 0.37630≤R≤0.37819。在这个范围内，任何值都可作为"IOU"的码字传送。特别地，二进制小数 .011000001 就在这个范围中，所以"IOU"可用 9 位传送（在这个例子中，霍夫曼码需用 10 位，参见第20.4节）。

当然，问题是要知道何时停止译码。小数 .011000001 不仅代表"IOU"，也可代表"IOU..."，其中省略号代表一个后继无限的字符串。为解决这个模棱两可的问题，算术编码通常假定存在特殊的第 $N_{ch}+1$ 个字符——EOM（消息的结尾），它只有在输入的末尾出现。由于 *EOM* 出现的概率很小，它在数据线上只分配了非常小的一段。




从最初的0至1区间对连续的字符不断地细分，最后数值就可用任何期望的基数表示成小数数字，这就是码字的输出。注意，每个字符分配的子区间正比于它出现的概率。

图20.5.1　虚构的语言 Voweilish 中消息"IOU..."的算术编码

在以上例子中，我们把 R 作为二进制小数给出。我们也可以把它以任何其它的基数输出，例如基数 94 或基数 36，只要方便于预期的存储或通信信道即可。

你可能会疑惑，对于一个长消息来说，如何处理这种似乎难以置信的对 $R$ 的精度要求。答案是，实际上 $R$ 不会一次完全表示出来。在任何给定阶段中，我们都有 $R$ 的上边界和下边界，它用输出基数表示成一个有限位的数。当上界数和下界数变成同一时，我们可以把它左移掉，并带进最低有效端的新的位。下面的程序有一个参数 NWK，它是保留工作位数目的参数。它必须足够大，以使退化偶然出现的机会接近于零。（如果一旦出现退化，本程序会给出信号。）由于放弃旧位和带进新位是一致地在编码和解码中进行的，所以要保持同步。

程序 **arcmak** 是构造累积频率分布表，它用以在每一阶段中分割区间。在源程序 **arcode**

中，当一个大小为 jdif 的区间按 n 正比于 ntot 来进行分割，则我们必须计算(n · jdif)/ntot。用整数运算，这个数可能会溢出。不幸的是，像 jdif/(ntot/n)这样的表达式是不相等的。在以下实现中，我们采用双精度的浮点运算来进行计算。这不仅没有效，而且因为在编码和解码过程中会出现同一舍入误差，尽管任何机器都是严格按编码方式来解码，但是不同的舍入误差(虽很少见)会使不同机器的编码有所不同。为了严格地使用，就需在一个双寄存器中，用整数计算(该计算在 C 程序中做不到)来替换这种浮点运算。

内设变量 minint 作为上、下界之间允许的最小不连续步长，它决定着何时加入新的低有效位。minint 必须足够大，以能够分辨所有输入的字符。也就是说，对所有 $i$ 我们必须有 $p_i \times \text{minint} > 1$。无论怎么大，$100N_{ch}$或 $1.1/\min p_i$ 一般已足够了。但是，为了安全起见，以下程序让 minint 尽可能地大，使 minint * nradd 的积刚好小于溢出。有时，这个结果不是有效的，在消息的结尾会输出一些不必要的字符。如果想靠近边缘，就可以减小 minint。

**arcmak** 中最后一个安全特性是，它对表 nfreq 中的零值象 1 一样进行处理。如果不这样，nfreq 入口为零的单个字符消息会搅乱剩下的整个消息。如果读者想要稍微更有效地编码而愿冒险的话，则可以删除 IMAX( ,1)操作。

```
#include "nrutil.h"
#include <limits.h>                           包括整数范围的 ANSI 标题文件
#define MC 512                                nchh 的最大预期值
#ifdef ULONG_MAX                              无符号长型的最大数
#define MAXINT (ULONG_MAX >> 1)               不会溢出最大正整数
#else
#define MAXINT 2147483647
#endif

typedef struct {
    unsigned long *ilob, *iupb, *ncumfq, jdif, nc, minint, nch, ncum, nrad;
} arithcode;

void arcmak(unsigned long nfreq[], unsigned long nchh, unsigned long nradd, arithcode *acode)
  给定 nchh 个符号的发生频率表 nfreq[1..nchh]，并给定一个期望的输出基数 nradd，初始化结构 acode 中累积频率
  表和其它变量。
{
    unsigned long j;

    if (nchh > MC) nrerror("input radix may not exceed MC in arcmak.");
    if (nradd > 256) nrerror("output radix may not exceed 256 in arcmak.");

    acode->minint=MAXINT/nradd;
    acode->nch=nchh;
    acode->nrad=nradd;
    acode->ncumfq[1]=0;
    for (j=2;j<=acode->nch+1;j++)
        acode->ncumfq[j]=acode->ncumfq[j-1]+IMAX(nfreq[j-1],1);
    acode->ncum=acode->ncumfq[acode->nch+2]=acode->ncumfq[acode->nch+1]+1;
}
```

必须用类似以下语句，在用户的主程序中定义和分配结构 acode：

```
#include "nrutil.h"
#define MC 512                       在 arcmak 中 nchh 的最大预期值
#define NWK 20                       保持此值与程序 arcode 中相同
typedef struct {
```

```
    unsigned long *ilob,*iupb,*ncumfq,jdif,nc,minint,nch,ncum,nrad;
} arithcode;
    ...
    arithcode acode;
    ...
    acode.ilob=(unsigned long *)lvector(1,NWK);          为 acode 分配空间
    acode.iupb=(unsigned long *)lvector(1,NWK);
    acode.ncumfq=(unsigned long *)lvector(1,MC+2);
```

采用下面程序 **arcode**，对消息中单个字符进行编码和译码，并它调用程序 **arcsum**。

```
#include <stdio.h>
#include <stdlib.h>
#define NWK 20
#define JTRY(j,k,m) ((long)((((double)(k))*((double)(j)))/((double)(m))))
    该宏用于计算无溢出时的(k*j)/m。若用一个执行双精度整数相乘的汇编语言的例程来替换之，本程序的有效性可
    得到改进

typedef struct {
    unsigned long *ilob,*iupb,*ncumfq,jdif,nc,minint,nch,ncum,nrad;
} arithcode;

void arcode(unsigned long *ich, unsigned char **codep, unsigned long *lcode,
    unsigned long *lcd, int isign, arithcode *acode)
    编码(isign=1)或译码(isign=-1)字符数组*codep[1..lcode]中的单个字符ich，以字节*codep[lcd]开始，并
    (若需要)增加lcd，以便指针lcd指向或返回到*codep中第一个不用的字节。注意，结构acode既包含码字信息，也
    包含写入数值*codep中的特殊输出的状态信息。无论对于编码或译码，对任意数组*codep开始前，需要用isign=
    0进行初始化调用。除初始化调用arcmak外，要求arcmak用于初始化码本身。若用ich=nch(在arcmak中设置)调
    用，预定表示"消息结束"
{
    void arcsum(unsigned long iin[], unsigned long iout[], unsigned long ja,
        int nwk, unsigned long nrad, unsigned long nc);
    void nrerror(char error_text[]);
    int j,k;
    unsigned long ihi,ja,jh,jl,m;

    if (!isign) {                   初始化上、下限的足够数字
        acode->jdif=acode->nrad-1;
        for (j=NWK;j>=1;j--) {
            acode->iupb[j]=acode->nrad-1;
            acode->ilob[j]=0;
            acode->nc=j;
            if (acode->jdif > acode->minint) return;     初始化结束
            acode->jdif=(acode->jdif+1)*acode->nrad-1;
        }
        nrerror("NWK too small in arcode.");
    } else {
        if (isign > 0) {            若编码，检验有效的输入字符
            if (*ich > acode->nch) nrerror("bad ich in arcode.");
        }
        else {                      若译码，用二分法查找字符ich
            ja=(*codep)[*lcd]-acode->ilob[acode->nc];
            for (j=acode->nc+1;j<=NWK;j++) {
                ja *= acode->nrad;
                ja += ((*codep)[*lcd+j-acode->nc]-acode->ilob[j]);
            }
            ihi=acode->nch+1;
            *ich=0;
            while (ihi-(*ich) > 1) {
                m=(*ich+ihi)>>1;
                if (ja >= JTRY(acode->jdif,acode->ncumfq[m+1],acode->ncum))
```

```
                *ich=m;
            else ihi=m;
        }
        if (*ich == acode->nch) return;        检测出消息结束
    }
    以下码对编码和译码是公用的。把字符ich转换成新的子区域 [ilob,iupb)
    jh=JTRY(acode->jdif,acode->ncumfq[*ich+2],acode->ncum);
    jl=JTRY(acode->jdif,acode->ncumfq[*ich+1],acode->ncum);
    acode->jdif=jh-jl;
    arcsum(acode->ilob,acode->iupb,jh,NWK,acode->nrad,acode->nc);
    arcsum(acode->ilob,acode->ilob,jl,NWK,acode->nrad,acode->nc);
    How many leading digits to output (if encoding) or skip over?
    for (j=acode->nc;j<=NWK;j++) {
        if (*ich != acode->nch && acode->iupb[j] != acode->ilob[j]) break;
        if (*lcd > *lcode) {
            fprintf(stderr,"Reached the end of the 'code' array.\n");
            fprintf(stderr,"Attempting to expand its size.\n");
            *lcode *= 1.5;
            if ((*codep=(unsigned char *)realloc(*codep,
                (unsigned)(*lcode*sizeof(unsigned char)))) == NULL) {
                nrerror("Size expansion failed");
            }
        }
        if (isign > 0) (*codep)[*lcd]=(unsigned char)acode->ilob[j];
        ++(*lcd);
    }
    if (j > NWK) return;      超出消息, 是否忘记对端点nch编码
    acode->nc=j;
    for(j=0;acode->jdif<acode->minint;j++)         多少数值要移位
        acode->jdif *= acode->nrad;
    if (acode->nc-j < 1) nrerror("NWK too small in arcode.");
    if (j) {                      对它们移位
        for (k=acode->nc;k<=NWK;k++) {
            acode->iupb[k-j]=acode->iupb[k];
            acode->ilob[k-j]=acode->ilob[k];
        }
    }
    acode->nc -= j;
    for (k=NWK-j+1;k<=NWK;k++) acode->iupb[k]=acode->ilob[k]=0;
  }
  return;                          正常返回
}



void arcsum(unsigned long iin[], unsigned long iout[], unsigned long ja,
    int nwk, unsigned long nrad, unsined long nc)
    用于 arcode,把整数 ja 加入到以基数 nrad 倍精度的整数 iin[nc..nwk]中。返回结果 iout[nc..nwk]。
{
    int j,karry=0;
    unsigned long jtmp;

    for (j=nwk;j>nc;j--) {
        jtmp=ja;
        ja /=nrad;
        iout[j]=iin[j]+(jtmp-ja*nrad)+karry;
        if (iout[j] >= nrad) {
            iout[j] -= nrad;
            karry=1;
        } else karry=0;
    }
    iout[nc]=iin[nc]+ja+karry;
}
```

若用户的主要目的是为了改变基数，而不是压缩（例如将任意文件转换成可打印的字符），则当然可以自由设置 nfreq 中的所有分量，如设为 1。

参考文献和进一步读物：

Bell, T. C., Cleary, J. G., and Witten, I. H. 1990, *Text Compression* (Englewood Cliffs, NJ: Prentice-Hall).

Nelson, M. 1991, *The Data Compression Book* (Redwood City, CA: M&T Books).

Witten, I. H., Neal, R. M., and Cleary, J. G. 1987, *Communications of the ACM*, vol. 30, pp. 520～540. [1]

## 20.6 任意精度的运算

让我们计算 $\pi$ 到小数点第 2 千位。为此，我们将学习计算机的一些关于多精度运算的问题，并将遇到非常不寻常的关于快速傅里叶变换(FFT)的应用。我们也将演变出一些可以用来在任何所要求的运算精度的水平上，进行计算的程序。

首先，我们需要一个 $\pi$ 的分析算法。有用的算法是二次收敛的，即每一次迭代使有效位数增加一倍。$\pi$ 二次收敛算法是基于 *AGM* 法（**算术几何平均法**，该方法也可应用于椭圆积分(见第6.11节)，以及椭圆偏微分方程(第19.5节)所采用的 ADI 法的先进实施。Borwein 和 Borwein[1]处理了这个课题，但它已超出了我们讨论的范围。他们对于 $\pi$ 的算法之一是，首先设置初始值为：

$$X_0 = \sqrt{2}$$
$$\pi_0 = 2 + \sqrt{2} \tag{20.6.1}$$
$$Y_0 = \sqrt[4]{2}$$

然后，当 $i=0,1,\cdots$ 重复迭代

$$X_{i+1} = \frac{1}{2}\left(\sqrt{X_i} + \frac{1}{\sqrt{X_i}}\right)$$
$$\pi_{i+1} = \pi_i\left(\frac{X_{i+1}+1}{Y_i+1}\right) \tag{20.6.2}$$
$$Y_{i+1} = \frac{Y_i\sqrt{X_{i+1}} + \frac{1}{\sqrt{X_{i+1}}}}{Y_i+1}$$

一直到出现 $\pi_\infty$。

现在，我们讨论怎样解决任意精度算法的问题：在高级语言中，如 C 语言，一个自然的选择就是以 256 为基数进行运算，因此，字符数组可直接译成数字串。一般在计算结束，总希望将结果转换成以 10 为基的数，因为人类的思维习惯于这种熟悉的语调，“叁点壹肆壹伍玖…”。然而对于计算机计算，我们可能永远离不开以256为基(或者通常可达到的 16 进制、8 进制或 2 进制)。

我们将采取存储数字串的习惯，并以人为的顺序进行存储，也就是说，数组中第一位存储的数字是最高有效位，最后一位存储的数字是最低有效位。当然，与此相反的习惯也可以

适用。"进位"，即我们需要将比 255 大的数划分成低位字节和高位进位，这就出现了一个编程的麻烦，在下面程序中采用宏 LOBYTE 和 HIBYTE 来解决这个问题。

这一点很简单，根据 Knuth[2]，编写一个"最快"算术运算的程序：捷加法(加一个单个字节到一串数中)、加法、减法、捷乘法(用一单个字节乘一串数)、捷除法、一次补码操作，以及一对实用的操作，复制和左移一串数(在磁盘中，这些函数都在一单独的 mpops.c 文件内)。

```
# define LOBYTE(x) ((unsigned char) ((x) & 0xff))
# define HIBYTE(x) ((unsigned char) ((x) >> 8 & 0xff))
    对译成以256为基数的字符串进行多精度算术运算，这组程序是搜集了简单的运算。

void mpadd(unsigned char w[], unsigned char u[], unsigned char v[], int n)
    无符号整数u[1..n]和v[1..n]相加，产生无符号整数w[1..n+1]，以256为基数。
{
    int j;
    unsigned short ireg=0;

    for (j=n;j>=1;j--) {
        ireg=u[j]+v[j]+HIBYTE(ireg);
        w[j+1]=LOBYTE(irge);
    }
    w[1]=HIBYTE(ireg);
}

void mpsub(int *is, unsigned char w[], unsigned char u[], unsigned char v[],int n)
    从以256为基数的无符号整数u[1..n]中减去v[1..n]，产生无符号整数w[1..n]。若相减结果为负，则is返回-1；
    反之，则返回0。
{
    int j;
    unsined short ireg=256;

    for (j=n;j>=1;j--) {
        ireg=255+u[j]-v[j]+HIBYTE(ireg);
        w[j]=LOBYTE(ireg);
    }
    *is=HIBYTE(ireg)-1;
}

void mpsad(unsigned char w[], unsinged char u[], int n, int iv)
    捷加法：将整数iv(0≤iv≤255)与以256为基数的无符号整数u[1..n]相加，产生w[1..n+1]。
{
    int j;
    unsigned short ireg;

    ireg=256*iv;
    for (j=n;j>=1;j--) {
        ireg=u[j]+HIBYTE(ireg);
        w[j+1]=LOBYTE(ireg);
    }
    w[1]=HIBYTE(ireg);
}

void mpsmu(unsigned char w[], unsinged char u[], int n, int iv)
    捷乘法：以256为基数的无符号整数u[1..n]乘以整数iv(0≤iv≤255)，产生w[1..n+1]。
{
    int j;
```

```
    unsigned short ireg=0;

    for (j=n;j>=1;j--) {
        ireg=u[j]*iv+HIBYTE(ireg);
        w[j+1]=LOBYTE(ireg);
    }
    w[1]=HIBYTE(ireg);
}

void mpsdv(unsigned char w[], unsinged char u[], int n, int iv, int *ir)
    短除法:以256为基数的无符号整数 u[1..n]被整数 iv 除(而0≤iv≤255),产生商为 w[1..n+1],和余数 ir(0≤ir≤
255)。
{
    int i,j;

    *ir=0;
    for (j=1;j<=n;j++) {
        i=256*(*ir)+u[j];
        w[j]=(unsigned char)(i/iv);
        *ir=i % iv;
    }
}

void mpneg(unsigned char u[], int n)
    对以256为基数的无符号整数 u[1..n]执行一次补码非操作。
{
    int j;
    unsigned short ireg=256;

    for (j=n;j>=1;j--) {
        ireg=255-u[j]+HIBYTE(ireg);
        u[j]=LOBYTE(ireg);
    }
}

void mpmov(unsigned char u[], unsinged char v[], int n)
    将 v[1..n]复制到 u[1..n]中
{
    int j;

    for (j=1;j<=n;j++) u[j]=v[j];
}

void mplsh (unsigned char u[], int n)
    将 u[2..n+1]左移成 u[1..n]。
{
    int j;

    for (j=1;j<=n;j++) u[j]=u[j+1];
}
```

两个数字串的全乘法,如果用传统的手算,不是快速的运算方法:两个长度均为 $N$ 的数字串相乘,被乘数将依此与乘数的每一个数字位作捷乘,共需 $O(N^2)$个操作。然而,我们将看到,对于长度为 $N$ 的数字的所有算术运算,实际上可按 $O(N\times\log N\times\log\log N)$个操作进行。

这种方法是,将乘法看作为乘数和被乘数的卷积,然后进行某种进位操作,例如考虑

下，456×789 的两种计算方法：

```
     456                        4  5  6
   × 789                      × 7  8  9
    4104                        36 45 54
   3648                      32 40 48
  3192                    28 35 42
  359784                  28 67 118 93 54
                         3  5  9  7  8  4
```

上图左边显示的是传统的乘法运算，该方法中，对全部被乘数（9、8、7）分别进行相乘，然后相加得到最后的结果。而图右边显示了不同的运算方法（有时用作心算），此时，所有单个数字进行相乘（例 8×6=48），然后按列相加得到没完成的进位结果（这里是 28、67、118、93、54）。最后一步是一个从右到左过程，记下一位低有效位数字，并且向左进位一个较高有效位数字，将它加到左边的总数中（例如，93+5=98，记录 8，进位 9）。

读者可立刻看出，在右边那种方法中，每一列的和是数字串卷积组成部分，例如 118=4×9+5×8+6×7。在第13.1节中，我们学过怎样用快速傅里叶变换（FFT）计算两个向量的卷积：每个向量作 FFT，然后将两个复变换式相乘，得到的结果是所求结果的逆 FFT。因为这个变换是采用了浮点算法，因此我们需要足够的精度，以使在出现舍入误差时，结果中每个组成部分的准确整数值仍是可辨认的。为了 FFT 的舍入误差，应该允许增加几倍 $\log_2(\log_2 N)$ 位二进位。以 256 为基数、长度为 N 字节的数可产生大到 $(256)^2N$ 阶的卷积分量，所以为了正确存储，需要 $16+\log_2 N$ 位精度。若数 it 是浮点尾数的二进位数（见第20.1节），则得条件：

$$16 + \log_2 N + 几个 \times \log_2\log_2 N < it \qquad (20.6.3)$$

我们看到，如果 it=24，则对于任意感兴趣的 $N$ 值，单精度是不合适的；如果 it=53，则双精度允许 $N$ 大于$10^6$，相当于几百万十进位数。所以下述程序假定有 **realft**（第12.3节）和 **four1**（第12.2节）的双精度版本，称之为 **drealft** 和 **dfour1**（这些程序包含在磁盘中）。

```
#include "nrutil.h"
#define RX 256.0

void mpmul(unsigned char w[], unsigned char u[],unsigned char v[], int n, int m)
    运用快速傅里叶变换计算以256为基数的无符号整数 u[1..n]与 v[1..m]相乘，产生积为 w[1..n+m]。
{
    void drealft(double data[], unsigned long n, int isign);     realft的双精度版本
    int j,mn,nn=1;
    double cy,t,*a,*b;

    mn=IMAX(m,n);
    while (nn < mn) nn <<= 1;          为变换找出最小可用的 2 的幂次
    nn <<= 1;
    a=dvector(1,nn);
    b=dvector(1,nn);
    for (j=1;j<=n;j++)                 将 u 移入双精度浮点型数组
        a[j]=(double)u[j];
    for (j=n+1;j<=nn;j++) a[j]=0.0;
    for (j=1;j<=m;j++)                 将 v 移入双精度浮点型数组
```

```
		b[j]=(double)v[j];
	for (j=m+1;j<=nn;j++) b[j]=0.0;
	drealft(a,nn,1);                    执行卷积：首先求出二个傅里叶变换
	drealft(b,nn,1);
	b[1] *= a[1];                       复数相乘的结果(实部和虚部)
	b[2] *= a[2];
	for (j=3;j<=nn;j+=2) {
		b[j]=(t=b[j])*a[j]-b[j+1]*a[j+1];
		b[j+1]=t*a[j+1]+b[j+1]*a[j];
	}
	drealft(b,nn,-1);                   进行傅氏反变换
	cy=0.0;                             执行最后完成所有进位的过程
	for (j=nn;j>=1;j--) {
		t=b[j]/(nn>>1)+cy+0.5;          这0.5是为了舍入误差而设计的
		b[j]=t;
		while (b[j] >= RX) b[j] -= RX;
		cy=(int) (t/RX);
	}
	if (cy >= RX) nrerror("cannot happen in fftmul");
	w[1]=(unsigned char) cy;            将结果复制输出
	for (j=2;j<=n+m;j++)
		w[j]=(unsigned char) b[j-1];
	free_dvector(b,1,nn);
	free_dvector(a,1,nn);
}
```

若采用这种"快速"乘法的操作过程，那么除法最好实现方式就是，用除数的倒数乘以被除数。倒数值 $V$ 用牛顿法则的迭代来计算：

$$U_{i+1} = U_i(2 - VU_i) \qquad (20.6.4)$$

这导致 $U_\infty$ 二次收敛于$1/V$，这一点的很容易证明(实际上，许多超级计算机和 RISC 机都是用这种迭代法来实现除法的)。现在，我们就知道上面所提到的运算量 $N\log N\log\log N$ 是在何处产生：傅里叶变换中要 $N\log N$ 运算量，以及它用牛顿法则的迭代收敛需要另增加一个 $\log\log N$ 的因子。

```
#include "nrutil.h"
#define MF 4
#define BI (1.0/256)

void mpinv(unsigned char u[], unsigned char v[], int n, int m)
字符串 v[1..m]是以256为基数的数，其在v[1]后有基数的小数点；u[1..n]是设置为它的倒数的最高有效数字，在u[1]前有基数的小数点。
{
	void mpmov(unsigned char u[], unsigned char v[], int n);
	void mpmul(unsigned char w[], unsigned char u[], unsigned char v[], int n,
		int m);
	void mpneg(unsigned char u[], int n);
	unsigned char *rr,*s;
	int i,j,maxmn,mm;
	float fu,fv;

	maxmn=IMAX(n,m);
	rr=cvector(1,1+(maxmn<<1));
	s=cvector(1,maxmn);
	mm=IMIN(MF,m);
	fv=(float) v[mm];                   采用原来的浮点运算以得到一个初始近似值
	for (j=mm-1;j>=1;j--) {
		fv *= BI;
		fv += v[j];
	}
	fu=1.0/fv;
	for (j=1;j<=n;j++) {
		i=(int) fu;
```

```
        u[j]=(unsigned char) i;
        fu=256.0*(fu-i);
    }
    for (;;) {                              迭代牛顿法则到收敛
        mpmul(rr,u,v,n,m);                  在S中构造2 − UV
        mpmov(s,&rr[1],n);
        mpneg(s,n);
        s[1] -= 254;                        将SU的乘积赋予U
        mpmul(rr,s,u,n,n);
        mpmov(u,&rr[1],n);
        for (j=2;j<n;j++)                   若S的小数部分不是零，它就没有收敛到1
            if (s[j]) break;
        if (j==n) {
            free_cvector(s,1,maxmn);
            free_cvector(rr,1,1+(maxmn<<1));
            return;
        }
    }
}
```

现在，除法只是一个简单的推论，仅需计算满足一定精度的倒数，从而得到商和余数。

```
#include "nrutil.h"
#define MACC 3

void mpdiv(unsigned char q[], unsigned char r[],unsigned char u[],
    unsigned char v[], int n, int m)
    以256为基数的无符号整数u[1..n]被v[1..m]除(要求m≤n)，产生商为q[1..n−m+1]和余数r[1..m]。
{
    void mpinv(unsigned char u[], unsigned char v[], int n, int m);
    void mpmov(unsigned char u[], unsigned char v[], int n);
    void mpmul(unsigned char w[], unsigned char u[], unsigned char v[], int n,
        int m);
    void mpsub(int *is, unsigned char w[], unsigned char u[], unsigned char v[],
        int n);
    int is;
    unsigned char *rr,*s;

    rr=cvector(1,(n+MACC)<<1);
    s=cvector(1,n+MACC);
    mpinv(s,v,n-m+MACC,m);                  设 S = 1/V.
    mpmul(rr,s,u,n-m+MACC,n);               设 Q = SU.
    mpmov(q,&rr[1],n-m+1);
    mpmul(rr,q,v,n-m+1,m);                  相乘和相减得到余数
    mpsub(&is,&rr[1],u,&rr[1],n);
    if (is) nrerror("MACC too small in mpdiv");
    mpmov(r,&rr[n-m+1],m);
    free_cvector(s,1,n+MACC);
    free_cvector(rr,1,(n+MACC)<<1);
}
```

用牛顿法则计算平方根和除法类似。若

$$U_{i+1} = \frac{1}{2}U_i(3 - VU_i^2) \qquad (20.6.5)$$

则$U_i$二次收敛于$1/\sqrt{V}$，最后用$V$除以得$\sqrt{V}$。

```
#include <math.h>
#include "nrutil.h"
#define MF 3
#define BI (1.0/256)
```

```
void mpsqrt(unsigned char w[], unsigned char u[], unsigned char v[], int n, int m)
    字符串 v[1..m]是以256为基数,其在v[1]后有基数小数点;w[1..n]设为v的平方根(基数小数点在w[1]之后)。在w
    和u不需要区别的情况下,它们可设为平方根。
{
    void mplsh(unsigned char u[], int n);
    void mpmov(unsigned char u[], unsigned char v[], int n);
    void mpmul(unsigned char w[], unsigned char u[], unsigned char v[], int n,
        int m);
    void mpneg(unsigned char u[], int n);
    void mpsdv(unsigned char w[], unsigned char u[], int n, int iv, int *ir);
    int i,ir,j,mm;
    float fu,fv;
    unsigned char *r,*s;

    r=cvector(1,n<<1);
    s=cvector(1,n<<1);
    mm=IMIN(m,MF);
    fv=(float) v[mm];                      采用原来浮点运算得到一个初始近似值
    for (j=mm-1;j>=1;j--) {
        fv *= BI;
        fv += v[j];
    }
    fu=1.0/sqrt(fv);
    for (j=1;j<=n;j++) {
        i=(int) fu;
        u[j]=(unsigned char) i;
        fu=256.0*(fu-i);
    }
    for (;;) {                                迭代牛顿法则以收敛
        mpmul(r,u,u,n,n);                     构造 S = (3 - VU^2)/2
        mplsh(r,n);
        mpmul(s,r,v,n,m);
        mplsh(s,n);
        mpneg(s,n);
        s[1] -= 253;
        mpsdv(s,s,n,2,&ir);
        for (j=2;j<n;j++) {                   若S的小数部分不为零,它不能收敛到1
            if (s[j]) {
                mpmul(r,s,u,n,n);             用SU替换U
                mpmov(u,&r[1],n);
                break;
            }
        }
        if (j<n) continue;
        mpmul(r,u,v,n,m);                     从倒数求得平方根并返回
        mpmov(w,&r[1],n);
        free_cvector(s,1,n<<1);
        free_cvector(r,1,n<<1);
        return;
    }
}
```

我们已经提到基数转换成十进制数只是人们的习惯问题,所以通常可以忽略。将一个小数转换在十进制的最简单的方法是反复乘以10,拾取(并减去)结果中的整数部分。然而,因为每次释出十进位数需$O(N)$次运算,所以这个过程共需$O(N^2)$运算量。基数转换也可以采用"分开并分别处理"的策略实现快速运算,其中小数用10的大幂次项来相乘,足以使所求数字的一半移到基数小数点的左边,现在,再分别处理整数和小数部分,各部分作进一步划分。如果我们要计算$\pi$的几十亿位数字,而不是几千位数字,则需实现这个方案。对于目前情况,下面较慢的程序是合适的。

```
#define IAZ 48

void mp2dfr(unsigned char a[], unsigned char s[], int n, int *m)
```
将以256为基数的小数a[1..n](基数小数点在a[1]之前)转换成十进位小数，它用ASCII字符串s[1..m]表示，其中m处返回值。输入数组a[1..n]将被破坏。注意：为了简便，本程序实现的是慢速算法(正比于$N^2$)，基数转换的快速算法的确存在(正比于$N\log N$)，但较复杂。
```
{
    void mplsh(unsigned char u[], int n);
    void mpsmu(unsigned char w[], unsigned char u[], int n, int iv);
    int j;

    *m=(int) (2.408*n);
    for (j=1;j<=(*m);j++) {
        mpsmu(a,a,n,10);
        s[j]=a[1]+IAZ;
        mplsh(a,n);
    }
}
```

最后，我们得到实现式(20.6.1)和(20.6.2)的程序：

```
#include <stdio.h>
#include "nrutil.h"
#define IAOFF 48

void mppi(int n)
```
计算和打印π的前n位字节的多精度程序。
```
{
    void mp2dfr(unsigned char a[], unsigned char s[], int n, int *m);
    void mpadd(unsigned char w[], unsigned char u[], unsigned char v[], int n);
    void mpinv(unsigned char u[], unsigned char v[], int n, int m);
    void mplsh(unsigned char u[], int n);
    void mpmov(unsigned char u[], unsigned char v[], int n);
    void mpmul(unsigned char w[], unsigned char u[], unsigned char v[], int n,
        int m);
    void mpsdv(unsigned char w[], unsigned char u[], int n, int iv, int *ir);
    void mpsqrt(unsigned char w[], unsigned char u[], unsigned char v[], int n,
        int m);
    int ir,j,m;
    unsigned char mm,*x,*y,*sx,*sxi,*t,*s,*pi;

    x=cvector(1,n+1);
    y=cvector(1,n<<1);
    sx=cvector(1,n);
    sxi=cvector(1,n);
    t=cvector(1,n<<1);
    s=cvector(1,3*n);
    pi=cvector(1,n+1);
    t[1]=2;                              设 T = 2.
    for (j=2;j<=n;j++) t[j]=0;
    mpsqrt(x,x,t,n,n);                   设 X_0 = √2.
    mpadd(pi,t,x,n);                     设 π_0 = 2 + √2.
    mplsh(pi,n);
    mpsqrt(sx,sxi,x,n,n);                设 Y_0 = 2^{1/4}
    mpmov(y,sx,n);
    for (;;) {
        mpadd(x,sx,sxi,n);               设 X_{i+1} = (X_i^{1/2} + X_i^{-1/2})/2.
        mpsdv(x,&x[1],n,2,&ir);
        mpsqrt(sx,sxi,x,n,n);            计算临时值 T = Y_i X_{i+1}^{1/2} + X_{i+1}^{-1/2}
        mpmul(t,y,sx,n,n);
```

```
		mpadd(&t[1],&t[1],&sxi,n);
		x[1]++;                          $X_{i+1}$ 和 $Y_i$ 增加
		y[1]++;
		mpinv(s,y,n,n);                  设 $Y_{i+1} = T/(Y_i+1)$
		mpmul(y,&t[2],s,n,n);
		mplsh(y,n);
		mpmul(t,x,s,n,n)                 计算临时值 $T=(X_{i+1}+1)/(Y_i+1)$
		mm=t[2]-1;                       若 $T=1$ 则有收敛
		j=t[n+1]-mm;
		if (j > 1 || j < -1) {
			for (j=3;j<=n;j++) {
				if (t[j] != mm) {
					mpmul(s,pi,&t[1],n,n);      Set $\pi_{i+1} = T\pi_i$.
					mpmov(pi,&s[1],n);
					break;
				}
			}
			if (j <= n) continue
		}
		printf("pi=\n");
		s[1]=pi[1]+IAOFF;
		s[2]='.';
		m=mm;
		mp2dfr(&pi[1],&s[2],n-1,&m)      为打印转换成十进制数。注意：这程序简单，仅作
		                                  说明用，是慢速算法($\propto N^2$)，快速转换算法
		                                  ($\propto N\ln N$)存在，但复杂

		s[m+3]=0;
		printf(" %64s\n",&s[1]);
		free_cvector(pi,1,n+1);
		free_cvector(s,1,3*n);
		free_cvector(t,1,n<<1);
		free_cvector(sxi,1,n);
		free_cvector(sx,1,n);
		free_cvector(y,1,n<<1);
		free_cvector(x,1,n+1);
		return;
	}
}
```

图20.6.1给出当 $n=1000$ 时的计算结果。作为一个练习，读者可能喜欢采用以上程序，针对 Ramanujan 的著名恒等式[3]

$$\frac{1}{\pi}=\frac{\sqrt{8}}{9801}\sum_{n=0}^{\infty}\frac{(4n)!(1103+26390n)}{(n!396^n)^4} \tag{20.6.6}$$

的前 12 项来检验图20.6.1中前 100 位数字。还可以用这程序来证明数 $2^{512}+1$ 不是素数，它有因子 2424833 和 7455602825647884208337395736200454918783366342657（它们实际上是素数，剩下的素数因子是 $7.416\times10^{98}$）[4]。

参考文献和进一步读物：


Borwein, J. M., and Borwein, P. B. 1987, *Pi and the AGM: A Study in Analytic Number Theory and Computational Complexity* (New York: Wiley). [1]

Knuth, D. E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §4.3. [2]

Ramanujan, S. 1927, *Collected Papers of Srinivasa Ramanujan*, G. H. Hardy, P. V. Seshu Aiyar, and B. M. Wilson, eds. (Cambridge, U. K.: Cambridge University Press), pp. 23～39. [3]

Kolata, G. 1990, June 20, *The New York Times*. [4]

3.1415926535897932384626433832795028841971693993751058209749445923078164062
862089986280348253421170679821480865132823066470938446095505822317253594081
284811174502841027019385211055596446229489549303819644288109756659334461284
756482337867831652712019091456485669234603486104543266482133936072602491412
737245870066063155881748815209209628292540917153643678925903600113305305488
204665213841469519415116094330572703657595919530921861173819326117931051185
480744623799627495673518857527248912279381830119491298336733624406566430860
213949463952247371907021798609437027705392171762931767523846748184676694051
320005681271452635608277857713427577896091736371787214684409012249534301465
495853710507922796892589235420199561121290219608640344181598136297747713099
605187072113499999983729780499510597317328160963185950244594553469083026425
223082533446850352619311881710100031378387528865875332083814206171776691473
035982534904287554687311595628638823537875937519577818577805321712268066130
019278766111959092164201989380952572010654858632788659361533818279682303019
520353018529689957736225994138912497217752834791315155748572424541506959508
295331168617278558890750983817546374649393192550604009277016711390098488240
128583616035637076601047101819429555961989467678374494482553797747268471040
475346462080466842590694912933136770289891521047521620569660240580381501935
112533824300355876402474964732639141992726042699227967823547816360093417216
412199245863150302861829745557067498385054945885869269956909272107975093029
553211653449872027559602364806654991198818347977535663698074265425278625518
184175746728909777727938000816470600161452491921732172147723501414419735685
481613611573525521334757418494684385233239073941433345477624168625189835694
855620992192221842725502542568876717904946016534668049886272327917860857843
838279679766814541009538837863609506800642251252051173929848960841284886269
456042419652850222106611863067442786220391949450471237137869609563643719172
874677646575739624138908658326459958133904780275900994657640789512694683983
525957098258226205224894077267194782684826014769909026401363944374553050682
034962524517493996514314298091906592509372216964615157098583874105978859597
729754989301617539284681382686838689427741559918559252459539594310499725246
808459872736446958486538367362226260991246080512438843904512441365497627807
977156914359977001296160894416948685558484063534220722258284886481584560285

π 的前2398位十进位数，它由本节程序计算求得.

图20.6.1

# 参考书籍

搜集在此的参考书籍都具有广泛的用途，在本书中均不止一节引用过它们。一些更为专门的资料，因为仅仅在本书某一节中引用过，所以不再在此列出。

首先列出的少量专著，是关于数值方法和数值分析方面值得个人收藏的、十分有用的核心书籍。这些书也是我们喜欢作为手头必备的书籍。


Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, vol. 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York)

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America)

Ames, W.F. 1977, *Numerical Methods for Partial Differential Equations*, 2nd ed. (New York: Academic Press)

Bratley, P., Fox, B.L., and Schrage, E.L. 1983, *A Guide to Simulation* (New York: Springer-Verlag)

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall)

Delves, L.M., and Mohamed, J.L. 1985, *Computational Methods for Integral Equations* (Cambridge, U.K.: Cambridge University Press)

Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* (Englewood Cliffs, NJ: Prentice-Hall)

Gill, P.E., Murray, W., and Wright, M.H. 1991, *Numerical Linear Algebra and Optimization*, vol. 1 (Redwood City, CA: Addison-Wesley)

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press)

Oppenheim, A.V., and Schafer, R.W. 1989, *Discrete-Time Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall)

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill)

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley)

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag)

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag)


其次，我们列出大量收搜到的书籍。按我们的观点，在任何正式的图书馆里，有关数值方法、数值分析或计算方法一类内容中，应该收藏这些书。


Bevington, P.R. 1969, *Data Reduction and Error Analysis for the Physical Sciences* (New York: McGraw-Hill)

Bloomfield, P. 1976, *Fourier Analysis of Time Series – An Introduction* (New

York: Wiley)

Bowers, R.L., and Wilson, J.R. 1991, *Numerical Modeling in Applied Physics and Astrophysics* (Boston: Jones & Bartlett)

Brent, R.P. 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, NJ: Prentice-Hall)

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall)

Brownlee, K.A. 1965, *Statistical Theory and Methodology*, 2nd ed. (New York: Wiley)

Bunch, J.R., and Rose, D.J. (eds.) 1976, *Sparse Matrix Computations* (New York: Academic Press)

Canutq, C., Hussaini, M.Y., Quarteroni, A., and Zang, T.A. 1988, *Spectral Methods in Fluid Dynamics* (New York: Springer-Verlag)

Carnahan, B., Luther, H.A., and Wilkes, J.O. 1969, *Applied Numerical Methods* (New York: Wiley)

Champeney, D.C. 1973, *Fourier Transforms and Their Physical Applications* (New York: Academic Press)

Childers, D.G. (ed.) 1978, *Modern Spectrum Analysis* (New York: IEEE Press)

Cooper, L., and Steinberg, D. 1970, *Introduction to Methods of Optimization* (Philadelphia: Saunders)

Dantzig, G.B. 1963, *Linear Programming and Extensions* (Princeton, NJ: Princeton University Press)

Devroye, L. 1986, *Non-Uniform Random Variate Generation* (New York: Springer-Verlag)

Dongarra, J.J., et al. 1979, *LINPACK User's Guide* (Philadelphia: S.I.A.M.)

Downie, N.M., and Heath, R.W. 1965, *Basic Statistical Methods*, 2nd ed. (New York: Harper & Row)

Duff, I.S., and Stewart, G.W. (eds.) 1979, *Sparse Matrix Proceedings 1978* (Philadelphia: S.I.A.M.)

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press)

Fike, C.T. 1968, *Computer Evaluation of Mathematical Functions* (Englewood Cliffs, NJ: Prentice-Hall)

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall)

Forsythe, G.E., and Moler, C.B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, NJ: Prentice-Hall)

Gass, S.T. 1969, *Linear Programming*, 3rd ed. (New York: McGraw-Hill)

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall)

Goodwin, E.T. (ed.) 1961, *Modern Computing Methods*, 2nd ed. (New York: Philosophical Library)

Gottlieb, D. and Orszag, S.A. 1977, *Numerical Analysis of Spectral Methods: Theory and Applications* (Philadelphia: S.I.A.M.)

Hackbusch, W. 1985, *Multi-Grid Methods and Applications* (New York: Springer-Verlag)

Hamming, R.W. 1962, *Numerical Methods for Engineers and Scientists*; reprinted 1986 (New York: Dover)

Hart, J.F., et al. 1968, *Computer Approximations* (New York: Wiley)

Hastings, C. 1955, *Approximations for Digital Computers* (Princeton: Princeton University Press)

Hildebrand, F.B. 1974, *Introduction to Numerical Analysis*, 2nd ed.; reprinted 1987 (New York: Dover)

Hoel, P.G. 1971, *Introduction to Mathematical Statistics*, 4th ed. (New York: Wiley)

Horn, R.A., and Johnson, C.R. 1985, *Matrix Analysis* (Cambridge: Cambridge University Press)

Householder, A.S. 1970, *The Numerical Treatment of a Single Nonlinear Equation* (New York: McGraw-Hill)

Huber, P.J. 1981, *Robust Statistics* (New York: Wiley)

Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods* (New York: Wiley)

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press)

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley)

Kahaner, D., Moler, C., and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs, NJ: Prentice Hall)

Keller, H.B. 1968, *Numerical Methods for Two-Point Boundary-Value Problems* (Waltham, MA: Blaisdell)

Knuth, D.E. 1968, *Fundamental Algorithms*, vol. 1 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley)

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley)

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley)

Koonin, S.E., and Meredith, D.C. 1990, *Computational Physics, Fortran Version* (Redwood City, CA: Addison-Wesley)

Kuenzi, H.P., Tzschach, H.G., and Zehnder, C.A. 1971, *Numerical Methods of Mathematical Optimization* (New York: Academic Press)

Lanczos, C. 1956, *Applied Analysis*; reprinted 1988 (New York: Dover)

Land, A.H., and Powell, S. 1973, *Fortran Codes for Mathematical Programming* (London: Wiley-Interscience)

Lawson, C.L., and Hanson, R. 1974, *Solving Least Squares Problems* (Englewood Cliffs, NJ: Prentice-Hall)

Lehmann, E.L. 1975, *Nonparametrics: Statistical Methods Based on Ranks* (San Francisco: Holden-Day)

Luke, Y.L. 1975, *Mathematical Functions and Their Approximations* (New York: Academic Press)

Magnus, W., and Oberhettinger, F. 1949, *Formulas and Theorems for the Functions of Mathematical Physics* (New York: Chelsea)

Martin, B.R. 1971, *Statistics for Physicists* (New York: Academic Press)

Mathews, J., and Walker, R.L. 1970, *Mathematical Methods of Physics*, 2nd ed. (Reading, MA: W.A. Benjamin/Addison-Wesley)

von Mises, R. 1964, *Mathematical Theory of Probability and Statistics* (New York: Academic Press)

Murty, K.G. 1976, *Linear and Combinatorial Programming* (New York: Wiley)

Norusis, M.J. 1982, *SPSS Introductory Guide: Basic Statistics and Operations*; and 1985, *SPSS-X Advanced Statistics Guide* (New York: McGraw-Hill)

Nussbaumer, H.J. 1982, *Fast Fourier Transform and Convolution Algorithms* (New York: Springer-Verlag)

Ortega, J., and Rheinboldt, W. 1970, *Iterative Solution of Nonlinear Equations in Several Variables* (New York: Academic Press)

Ostrowski, A.M. 1966, *Solutions of Equations and Systems of Equations*, 2nd ed. (New York: Academic Press)

Polak, E. 1971, *Computational Methods in Optimization* (New York: Academic Press)

Rice, J.R. 1983, *Numerical Methods, Software, and Analysis* (New York: McGraw-Hill)

Richtmyer, R.D., and Morton, K.W. 1967, *Difference Methods for Initial Value Problems*, 2nd ed. (New York: Wiley-Interscience)

Roache, P.J. 1976, *Computational Fluid Dynamics* (Albuquerque: Hermosa)

Robinson, E.A., and Treitel, S. 1980, *Geophysical Signal Analysis* (Englewood Cliffs, NJ: Prentice-Hall)

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag)

Stuart, A., and Ord, J.K. 1987, *Kendall's Advanced Theory of Statistics*, 5th ed. (London: Griffin and Co.) [previous eds. published as Kendall, M., and Stuart, A., *The Advanced Theory of Statistics*]

Tewarson, R.P. 1973, *Sparse Matrices* (New York: Academic Press)

Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley)

Wilkinson, J.H. 1965, *The Algebraic Eigenvalue Problem* (New York: Oxford University Press)

Young, D.M., and Gregory, R.T. 1973, *A Survey of Numerical Mathematics*, 2 vols.; reprinted 1988 (New York: Dover)

# 附录 A　原型说明表

在此，我们列出了本书所有程序的 ANSI 标准的原型说明。正如所建议的，如果读者使用的编译器是基于 ANSI 标准的，那么，当包含或引用本书中任何程序的源文件进行编译时，就只需在每个源文件中，用♯include 指令把本表 nr.h（或者相关的行）列入。它们会提醒用户的编译器，注意到我们的程序一般不需作变量换算，并且它们允许用户的编译器，指出程序调用中的可能错误。

本附录安装在磁盘的 nr.h 文件中。需重要指出的是，盘中同时包含 ANSI C 和“传统的”或“K&R”的两种类型说明。要调用 ANSI 的形式，可定义以下宏指令：__STDC__、ANSI 或者 NRANSI（最后一个宏指令是提供用户一种引用，它不会与前两个名字的其它可能应用相冲突）。如果读者拥有 ANSI 编译器，就一定要用这些定义的宏指令来调用标题文件。典型的方法就是，在编译器的命令行处加入一个开关，如“－DANSI”。

上述讨论不适用于使用“传统的”或“K&R”C 编译器的用户：不需要将 ANSI 的标题文件列入源文件。如果有 nr.h 的磁盘文件，无需设置任何宏指令__STDC__、ANSI 或 NRANSI，就可随意使用♯include nr.h。但必须提醒编译器，注意程序返回的数值类型。当然，必须保证是使用“K&R”版本的程序，这些程序也包含在《C 语言数值算法程序大全》的磁盘中。无论用户喜欢与否，编译器都将进行“通常的变量换算”，但要废除这种换算，需在程序中说明这些变量不同于通常的换算。

以下是 nr.h 文件清单：

```
#ifndef _NR_H_
#define _NR_H_

#ifndef _FCOMPLEX_DECLARE_T_
typedef struct FCOMPLEX {float r,i;} fcomplex;
#define _FCOMPLEX_DECLARE_T_
#endif /* _FCOMPLEX_DECLARE_T_ */

#ifndef _ARITHCODE_DECLARE_T_
typedef struct {
    unsigned long *ilob,*iupb,*ncumfq,jdif,nc,minint,nch,ncum,nrad;
} arithcode;
#define _ARITHCODE_DECLARE_T_
#endif /* _ARITHCODE_DECLARE_T_ */

#ifndef _HUFFCODE_DECLARE_T_
typedef struct {
    unsigned long *icod,*ncod,*left,*right,nch,nodemax;
} huffcode;
#define _HUFFCODE_DECLARE_T_
#endif /* _HUFFCODE_DECLARE_T_ */

#include <stdio.h>

#if defined(__STDC__) || defined(ANSI) || defined(NRANSI) /* ANSI */

void addint(double **uf, double **uc, double **res, int nf);
void airy(float x, float *ai, float *bi, float *aip, float *bip);
```

```
void amebsa(float **p, float y[], int ndim, float pb[],    float *yb,
    float ftol, float (*funk)(float []), int *iter, float temptr);
void amoeba(float **p, float y[], int ndim, float ftol,
    float (*funk)(float []), int *iter);
float amotry(float **p, float y[], float psum[], int ndim,
    float (*funk)(float []), int ihi, float fac);
float amotsa(float **p, float y[], float psum[], int ndim, float pb[],
    float *yb, float (*funk)(float []), int ihi, float *yhi, float fac);
void anneal(float x[], float y[], int iorder[], int ncity);
double anorm2(double **a, int n);
void arcmak(unsigned long nfreq[], unsigned long nchh, unsigned long nradd,
    arithcode *acode);
void arcode(unsigned long *ich, unsigned char **codep, unsigned long *lcode,
    unsigned long *lcd, int isign, arithcode *acode);
void arcsum(unsigned long iin[], unsigned long iout[], unsigned long ja,
    int nwk, unsigned long nrad, unsigned long nc);
void asolve(unsigned long n, double b[], double x[], int itrnsp);
void atimes(unsigned long n, double x[], double r[], int itrnsp);
void avevar(float data[], unsigned long n, float *ave, float *var);
void balanc(float **a, int n);
void banbks(float **a, unsigned long n, int m1, int m2, float **al,
    unsigned long indx[], float b[]);
void bandec(float **a, unsigned long n, int m1, int m2, float **al,
    unsigned long indx[], float *d);
void banmul(float **a, unsigned long n, int m1, int m2, float x[], float b[]);
void bcucof(float y[], float y1[], float y2[], float y12[], float d1,
    float d2, float **c);
void bcuint(float y[], float y1[], float y2[], float y12[],
    float x1l, float x1u, float x2l, float x2u, float x1,
    float x2, float *ansy, float *ansy1, float *ansy2);
void beschb(double x, double *gam1, double *gam2, double *gampl,
    double *gammi);
float bessi(int n, float x);
float bessi0(float x);
float bessi1(float x);
void bessik(float x, float xnu, float *ri, float *rk, float *rip,
    float *rkp);
float bessj(int n, float x);
float bessj0(float x);
float bessj1(float x);
void bessjy(float x, float xnu, float *rj, float *ry, float *rjp,
    float *ryp);
float bessk(int n, float x);
float bessk0(float x);
float bessk1(float x);
float bessy(int n, float x);
float bessy0(float x);
float bessy1(float x);
float beta(float z, float w);
float betacf(float a, float b, float x);
float betai(float a, float b, float x);
float bico(int n, int k);
void bksub(int ne, int nb, int jf, int k1, int k2, float ***c);
float bnldev(float pp, int n, long *idum);
float brent(float ax, float bx, float cx,
    float (*f)(float), float tol, float *xmin);
void broydn(float x[], int n, int *check,
    void (*vecfunc)(int, float [], float []));
void bsstep(float y[], float dydx[], int nv, float *xx, float htry,
    float eps, float yscal[], float *hdid, float *hnext,
    void (*derivs)(float, float [], float []));
void caldat(long julian, int *mm, int *id, int *iyyy);
void chder(float a, float b, float c[], float cder[], int n);
float chebev(float a, float b, float c[], int m, float x);
void chebft(float a, float b, float c[], int n, float (*func)(float));
void chebpc(float c[], float d[], int n);
void chint(float a, float b, float c[], float cint[], int n);
```

```
float chixy(float bang);
void choldc(float **a, int n, float p[]);
void cholsl(float **a, int n, float p[], float b[], float x[]);
void chsone(float bins[], float ebins[], int nbins, int knstrn,
    float *df, float *chsq, float *prob);
void chstwo(float bins1[], float bins2[], int nbins, int knstrn,
    float *df, float *chsq, float *prob);
void cisi(float x, float *ci, float *si);
void cntab1(int **nn, int ni, int nj, float *chisq,
    float *df, float *prob, float *cramrv, float *ccc);
void cntab2(int **nn, int ni, int nj, float *h, float *hx, float *hy,
    float *hygx, float *hxgy, float *uygx, float *uxgy, float *uxy);
void convlv(float data[], unsigned long n, float respns[], unsigned long m,
    int isign, float ans[]);
void copy(double **aout, double **ain, int n);
void correl(float data1[], float data2[], unsigned long n, float ans[]);
void cosft(float y[], int n, int isign);
void cosft1(float y[], int n);
void cosft2(float y[], int n, int isign);
void covsrt(float **covar, int ma, int ia[], int mfit);
void crank(unsigned long n, float w[], float *s);
void cyclic(float a[], float b[], float c[], float alpha, float beta,
    float r[], float x[], unsigned long n);
void daub4(float a[], unsigned long n, int isign);
float dawson(float x);
float dbrent(float ax, float bx, float cx,
    float (*f)(float), float (*df)(float), float tol, float *xmin);
void ddpoly(float c[], int nc, float x, float pd[], int nd);
int decchk(char string[], int n, char *ch);
void derivs(float x, float y[], float dydx[]);
float df1dim(float x);
void dfour1(double data[], unsigned long nn, int isign);
void dfpmin(float p[], int n, float gtol, int *iter, float *fret,
    float (*func)(float []), void (*dfunc)(float [], float []));
float dfridr(float (*func)(float), float x, float h, float *err);
void dftcor(float w, float delta, float a, float b, float endpts[],
    float *corre, float *corim, float *corfac);
void dftint(float (*func)(float), float a, float b, float w,
    float *cosint, float *sinint);
void difeq(int k, int k1, int k2, int jsf, int is1, int isf,
    int indexv[], int ne, float **s, float **y);
void dlinmin(float p[], float xi[], int n, float *fret,
    float (*func)(float []), void (*dfunc)(float [], float[]));
double dpythag(double a, double b);
void drealft(double data[], unsigned long n, int isign);
void dsprsax(double sa[], unsigned long ijs[], double x[], double b[],
    unsigned long n);
void dsprstx(double sa[], unsigned long ija[], double x[], double b[],
    unsigned long n);
void dsvbksb(double **u, double w[], double **v, int m, int n, double b[],
    double x[]);
void dsvdcmp(double **a, int m, int n, double w[], double **v);
void eclass(int nf[], int n, int lista[], int listb[], int m);
void eclazz(int nf[], int n, int (*equiv)(int, int));
float ei(float x);
void eigsrt(float d[], float **v, int n);
float elle(float phi, float ak);
float ellf(float phi, float ak);
float ellpi(float phi, float en, float ak);
void elmhes(float **a, int n);
float erfcc(float x);
float erff(float x);
float erffc(float x);
void eulsum(float *sum, float term, int jterm, float wksp[]);
float evlmem(float fdt, float d[], int m, float xms);
float expdev(long *idum);
float expint(int n, float x);
```

```
float f1(float x);
float f1dim(float x);
float f2(float y);
float f3(float z);
float factln(int n);
float factrl(int n);
void fasper(float x[], float y[], unsigned long n, float ofac, float hifac,
    float wk1[], float wk2[], unsigned long nwk, unsigned long *nout,
    unsigned long *jmax, float *prob);
void fdjac(int n, float x[], float fvec[], float **df,
    void (*vecfunc)(int, float [], float []));
void fgauss(float x, float a[], float *y, float dyda[], int na);
void fill0(double **u, int n);
void fit(float x[], float y[], int ndata, float sig[], int mwt,
    float *a, float *b, float *siga, float *sigb, float *chi2, float *q);
void fitexy(float x[], float y[], int ndat, float sigx[], float sigy[],
    float *a, float *b, float *siga, float *sigb, float *chi2, float *q);
void fixrts(float d[], int m);
void fleg(float x, float pl[], int nl);
void flmoon(int n, int nph, long *jd, float *frac);
float fmin(float x[]);
void four1(float data[], unsigned long nn, int isign);
void fourew(FILE *file[5], int *na, int *nb, int *nc, int *nd);
void fourfs(FILE *file[5], unsigned long nn[], int ndim, int isign);
void fourn(float data[], unsigned long nn[], int ndim, int isign);
void fpoly(float x, float p[], int np);
void fred2(int n, float a, float b, float t[], float f[], float w[],
    float (*g)(float), float (*ak)(float, float));
float fredin(float x, int n, float a, float b, float t[], float f[], float w[],
    float (*g)(float), float (*ak)(float, float));
void frenel(float x, float *s, float *c);
void frprmn(float p[], int n, float ftol, int *iter, float *fret,
    float (*func)(float []), void (*dfunc)(float [], float []));
void ftest(float data1[], unsigned long n1, float data2[], unsigned long n2,
    float *f, float *prob);
float gamdev(int ia, long *idum);
float gammln(float xx);
float gammp(float a, float x);
float gammq(float a, float x);
float gasdev(long *idum);
void gaucof(int n, float a[], float b[], float amu0, float x[], float w[]);
void gauher(float x[], float w[], int n);
void gaujac(float x[], float w[], int n, float alf, float bet);
void gaulag(float x[], float w[], int n, float alf);
void gauleg(float x1, float x2, float x[], float w[], int n);
void gaussj(float **a, int n, float **b, int m);
void gcf(float *gammcf, float a, float x, float *gln);
float golden(float ax, float bx, float cx, float (*f)(float), float tol,
    float *xmin);
void gser(float *gamser, float a, float x, float *gln);
void hpsel(unsigned long m, unsigned long n, float arr[], float heap[]);
void hpsort(unsigned long n, float ra[]);
void hqr(float **a, int n, float wr[], float wi[]);
void hufapp(unsigned long index[], unsigned long nprob[], unsigned long n,
    unsigned long i);
void hufdec(unsigned long *ich, unsigned char *code, unsigned long lcode,
    unsigned long *nb, huffcode *hcode);
void hufenc(unsigned long ich, unsigned char **codep, unsigned long *lcode,
    unsigned long *nb, huffcode *hcode);
void hufmak(unsigned long nfreq[], unsigned long nchin, unsigned long *ilong,
    unsigned long *nlong, huffcode *hcode);
void hunt(float xx[], unsigned long n, float x, unsigned long *jlo);
void hypdrv(float s, float yy[], float dyyds[]);
fcomplex hypgeo(fcomplex a, fcomplex b, fcomplex c, fcomplex z);
void hypser(fcomplex a, fcomplex b, fcomplex c, fcomplex z,
    fcomplex *series, fcomplex *deriv);
```

```
unsigned short icrc(unsigned short crc, unsigned char *bufptr,
    unsigned long len, short jinit, int jrev);
unsigned short icrc1(unsigned short crc, unsigned char onech);
unsigned long igray(unsigned long n, int is);
void iindexx(unsigned long n, long arr[], unsigned long indx[]);
void indexx(unsigned long n, float arr[], unsigned long indx[]);
void interp(double **uf, double **uc, int nf);
int irbit1(unsigned long *iseed);
int irbit2(unsigned long *iseed);
void jacobi(float **a, int n, float d[], float **v, int *nrot);
void jacobn(float x, float y[], float dfdx[], float **dfdy, int n);
long julday(int mm, int id, int iyyy);
void kendl1(float data1[], float data2[], unsigned long n, float *tau, float *z,
    float *prob);
void kendl2(float **tab, int i, int j, float *tau, float *z, float *prob);
void kermom(double w[], double y, int m);
void ks2d1s(float x1[], float y1[], unsigned long n1,
    void (*quadvl)(float, float, float *, float *, float *, float *),
    float *d1, float *prob);
void ks2d2s(float x1[], float y1[], unsigned long n1, float x2[], float y2[],
    unsigned long n2, float *d, float *prob);
void ksone(float data[], unsigned long n, float (*func)(float), float *d,
    float *prob);
void kstwo(float data1[], unsigned long n1, float data2[], unsigned long n2,
    float *d, float *prob);
void laguer(fcomplex a[], int m, fcomplex *x, int *its);
void lfit(float x[], float y[], float sig[], int ndat, float a[], int ia[],
    int ma, float **covar, float *chisq, void (*funcs)(float, float [], int));
void linbcg(unsigned long n, double b[], double x[], int itol, double tol,
     int itmax, int *iter, double *err);
void linmin(float p[], float xi[], int n, float *fret,
    float (*func)(float []));
void lnsrch(int n, float xold[], float fold, float g[], float p[], float x[],
     float *f, float stpmax, int *check, float (*func)(float []));
void load(float x1, float v[], float y[]);
void load1(float x1, float v1[], float y[]);
void load2(float x2, float v2[], float y[]);
void locate(float xx[], unsigned long n, float x, unsigned long *j);
void lop(double **out, double **u, int n);
void lubksb(float **a, int n, int *indx, float b[]);
void ludcmp(float **a, int n, int *indx, float *d);
void machar(int *ibeta, int *it, int *irnd, int *ngrd,
    int *machep, int *negep, int *iexp, int *minexp, int *maxexp,
    float *eps, float *epsneg, float *xmin, float *xmax);
void matadd(double **a, double **b, double **c, int n);
void matsub(double **a, double **b, double **c, int n);
void medfit(float x[], float y[], int ndata, float *a, float *b, float *abdev);
void memcof(float data[], int n, int m, float *xms, float d[]);
int metrop(float de, float t);
void mgfas(double **u, int n, int maxcyc);
void mglin(double **u, int n, int ncycle);
float midexp(float (*funk)(float), float aa, float bb, int n);
float midinf(float (*funk)(float), float aa, float bb, int n);
float midpnt(float (*func)(float), float a, float b, int n);
float midsql(float (*funk)(float), float aa, float bb, int n);
float midsqu(float (*funk)(float), float aa, float bb, int n);
void miser(float (*func)(float []), float regn[], int ndim, unsigned long npts,
    float dith, float *ave, float *var);
void mmid(float y[], float dydx[], int nvar, float xs, float htot,
    int nstep, float yout[], void (*derivs)(float, float[], float[]));
void mnbrak(float *ax, float *bx, float *cx, float *fa, float *fb,
    float *fc, float (*func)(float));
void mnewt(int ntrial, float x[], int n, float tolx, float tolf);
void moment(float data[], int n, float *ave, float *adev, float *sdev,
    float *var, float *skew, float *curt);
void mp2dfr(unsigned char a[], unsigned char s[], int n, int *m);
```

```
void mpadd(unsigned char w[], unsigned char u[], unsigned char v[], int n);
void mpdiv(unsigned char q[], unsigned char r[], unsigned char u[],
    unsigned char v[], int n, int m);
void mpinv(unsigned char u[], unsigned char v[], int n, int m);
void mplsh(unsigned char u[], int n);
void mpmov(unsigned char u[], unsigned char v[], int n);
void mpmul(unsigned char w[], unsigned char u[], unsigned char v[], int n,
    int m);
void mpneg(unsigned char u[], int n);
void mppi(int n);
void mprove(float **a, float **alud, int n, int indx[], float b[],
    float x[]);
void mpsad(unsigned char w[], unsigned char u[], int n, int iv);
void mpsdv(unsigned char w[], unsigned char u[], int n, int iv, int *ir);
void mpsmu(unsigned char w[], unsigned char u[], int n, int iv);
void mpsqrt(unsigned char w[], unsigned char u[], unsigned char v[], int n,
    int m);
void mpsub(int *is, unsigned char w[], unsigned char u[], unsigned char v[],
    int n);
void mrqcof(float x[], float y[], float sig[], int ndata, float a[],
    int ia[], int ma, float **alpha, float beta[], float *chisq,
    void (*funcs)(float, float [], float *, float [], int));
void mrqmin(float x[], float y[], float sig[], int ndata, float a[],
    int ia[], int ma, float **covar, float **alpha, float *chisq,
    void (*funcs)(float, float [], float *, float [], int), float *alamda);
void newt(float x[], int n, int *check,
    void (*vecfunc)(int, float [], float []));
void odeint(float ystart[], int nvar, float x1, float x2,
    float eps, float h1, float hmin, int *nok, int *nbad,
    void (*derivs)(float, float [], float []),
    void (*rkqs)(float [], float [], int, float *, float, float,
    float [], float *, float *, void (*)(float, float [], float [])));
void orthog(int n, float anu[], float alpha[], float beta[], float a[],
    float b[]);
void pade(double cof[], int n, float *resid);
void pccheb(float d[], float c[], int n);
void pcshft(float a, float b, float d[], int n);
void pearsn(float x[], float y[], unsigned long n, float *r, float *prob,
    float *z);
void period(float x[], float y[], int n, float ofac, float hifac,
    float px[], float py[], int np, int *nout, int *jmax, float *prob);
void piksr2(int n, float arr[], float brr[]);
void piksrt(int n, float arr[]);
void pinvs(int ie1, int ie2, int je1, int jsf, int jc1, int k,
    float ***c, float **s);
float plgndr(int l, int m, float x);
float poidev(float xm, long *idum);
void polcoe(float x[], float y[], int n, float cof[]);
void polcof(float xa[], float ya[], int n, float cof[]);
void poldiv(float u[], int n, float v[], int nv, float q[], float r[]);
void polin2(float x1a[], float x2a[], float **ya, int m, int n,
    float x1, float x2, float *y, float *dy);
void polint(float xa[], float ya[], int n, float x, float *y, float *dy);
void powell(float p[], float **xi, int n, float ftol, int *iter, float *fret,
    float (*func)(float []));
void predic(float data[], int ndata, float d[], int m, float future[], int nfut);
float probks(float alam);
void psdes(unsigned long *lword, unsigned long *irword);
void pwt(float a[], unsigned long n, int isign);
void pwtset(int n);
float pythag(float a, float b);
void pzextr(int iest, float xest, float yest[], float yz[], float dy[],
    int nv);
float qgaus(float (*func)(float), float a, float b);
void qrdcmp(float **a, int n, float *c, float *d, int *sing);
float qromb(float (*func)(float), float a, float b);
float qromo(float (*func)(float), float a, float b,
```

```
    float (*choose)(float (*)(float), float, float, int));
void qroot(float p[], int n, float *b, float *c, float eps);
void qrsolv(float **a, int n, float c[], float d[], float b[]);
void qrupdt(float **r, float **qt, int n, float u[], float v[]);
float qsimp(float (*func)(float), float a, float b);
float qtrap(float (*func)(float), float a, float b);
float quad3d(float (*func)(float, float, float), float x1, float x2);
void quadct(float x, float y, float xx[], float yy[], unsigned long nn,
    float *fa, float *fb, float *fc, float *fd);
void quadmx(float **a, int n);
void quadvl(float x, float y, float *fa, float *fb, float *fc, float *fd);
float ran0(long *idum);
float ran1(long *idum);
float ran2(long *idum);
float ran3(long *idum);
float ran4(long *idum);
void rank(unsigned long n, unsigned long indx[], unsigned long irank[]);
void ranpt(float pt[], float regn[], int n);
void ratint(float xa[], float ya[], int n, float x, float *y, float *dy);
void ratlsq(double (*fn)(double), double a, double b, int mm, int kk,
    double cof[], double *dev);
double ratval(double x, double cof[], int mm, int kk);
float rc(float x, float y);
float rd(float x, float y, float z);
void realft(float data[], unsigned long n, int isign);
void rebin(float rc, int nd, float r[], float xin[], float xi[]);
void red(int iz1, int iz2, int jz1, int jz2, int jm1, int jm2, int jmf,
    int ic1, int jc1, int jcf, int kc, float ***c, float **s);
void relax(double **u, double **rhs, int n);
void relax2(double **u, double **rhs, int n);
void resid(double **res, double **u, double **rhs, int n);
float revcst(float x[], float y[], int iorder[], int ncity, int n[]);
void reverse(int iorder[], int ncity, int n[]);
float rf(float x, float y, float z);
float rj(float x, float y, float z, float p);
void rk4(float y[], float dydx[], int n, float x, float h, float yout[],
    void (*derivs)(float, float [], float []));
void rkck(float y[], float dydx[], int n, float x, float h,
    float yout[], float yerr[], void (*derivs)(float, float [], float []));
void rkdumb(float vstart[], int nvar, float x1, float x2, int nstep,
    void (*derivs)(float, float [], float []));
void rkqs(float y[], float dydx[], int n, float *x,
    float htry, float eps, float yscal[], float *hdid, float *hnext,
    void (*derivs)(float, float [], float []));
void rlft3(float ***data, float **speq, unsigned long nn1,
    unsigned long nn2, unsigned long nn3, int isign);
float rofunc(float b);
void rotate(float **r, float **qt, int n, int i, float a, float b);
void rsolv(float **a, int n, float d[], float b[]);
void rstrct(double **uc, double **uf, int nc);
float rtbis(float (*func)(float), float x1, float x2, float xacc);
float rtflsp(float (*func)(float), float x1, float x2, float xacc);
float rtnewt(void (*funcd)(float, float *, float *), float x1, float x2,
    float xacc);
float rtsafe(void (*funcd)(float, float *, float *), float x1, float x2,
    float xacc);
float rtsec(float (*func)(float), float x1, float x2, float xacc);
void rzextr(int iest, float xest, float yest[], float yz[], float dy[], int nv);
void savgol(float c[], int np, int nl, int nr, int ld, int m);
void score(float xf, float y[], float f[]);
void scrsho(float (*fx)(float));
float select(unsigned long k, unsigned long n, float arr[]);
float selip(unsigned long k, unsigned long n, float arr[]);
void shell(unsigned long n, float a[]);
void shoot(int n, float v[], float f[]);
void shootf(int n, float v[], float f[]);
void simp1(float **a, int mm, int ll[], int nll, int iabf, int *kp,
```

```
    float *bmax);
void simp2(float **a, int n, int l2[], int nl2, int *ip, int kp, float *q1);
void simp3(float **a, int i1, int k1, int ip, int kp);
void simplx(float **a, int m, int n, int m1, int m2, int m3, int *icase,
    int izrov[], int iposv[]);
void simpr(float y[], float dydx[], float dfdx[], float **dfdy,
    int n, float xs, float htot, int nstep, float yout[],
    void (*derivs)(float, float [], float []));
void sinft(float y[], int n);
void slvsm2(double **u, double **rhs);
void slvsml(double **u, double **rhs);
void sncndn(float uu, float emmc, float *sn, float *cn, float *dn);
double snrm(unsigned long n, double sx[], int itol);
void sobseq(int *n, float x[]);
void solvde(int itmax, float conv, float slowc, float scalv[],
    int indexv[], int ne, int nb, int m, float **y, float ***c, float **s);
void sor(double **a, double **b, double **c, double **d, double **e,
    double **f, double **u, int jmax, double rjac);
void sort(unsigned long n, float arr[]);
void sort2(unsigned long n, float arr[], float brr[]);
void sort3(unsigned long n, float ra[], float rb[], float rc[]);
void spctrm(FILE *fp, float p[], int m, int k, int ovrlap);
void spear(float data1[], float data2[], unsigned long n, float *d, float *zd,
    float *probd, float *rs, float *probrs);
void sphbes(int n, float x, float *sj, float *sy, float *sjp, float *syp);
void splie2(float x1a[], float x2a[], float **ya, int m, int n, float **y2a);
void splin2(float x1a[], float x2a[], float **ya, float **y2a, int m, int n,
    float x1, float x2, float *y);
void spline(float x[], float y[], int n, float yp1, float ypn, float y2[]);
void splint(float xa[], float ya[], float y2a[], int n, float x, float *y);
void spread(float y, float yy[], unsigned long n, float x, int m);
void sprsax(float sa[], unsigned long ija[], float x[], float b[],
    unsigned long n);
void sprsin(float **a, int n, float thresh, unsigned long nmax, float sa[],
    unsigned long ija[]);
void sprspm(float sa[], unsigned long ija[], float sb[], unsigned long ijb[],
    float sc[], unsigned long ijc[]);
void sprstm(float sa[], unsigned long ija[], float sb[], unsigned long ijb[],
    float thresh, unsigned long nmax, float sc[], unsigned long ijc[]);
void sprstp(float sa[], unsigned long ija[], float sb[], unsigned long ijb[]);
void sprstx(float sa[], unsigned long ija[], float x[], float b[],
    unsigned long n);
void stifbs(float y[], float dydx[], int nv, float *xx,
    float htry, float eps, float yscal[], float *hdid, float *hnext,
    void (*derivs)(float, float [], float []));
void stiff(float y[], float dydx[], int n, float *x,
    float htry, float eps, float yscal[], float *hdid, float *hnext,
    void (*derivs)(float, float [], float []));
void stoerm(float y[], float d2y[], int nv, float xs,
    float htot, int nstep, float yout[],
    void (*derivs)(float, float [], float []));
void svbksb(float **u, float w[], float **v, int m, int n, float b[],
    float x[]);
void svdcmp(float **a, int m, int n, float w[], float **v);
void svdfit(float x[], float y[], float sig[], int ndata, float a[],
    int ma, float **u, float **v, float w[], float *chisq,
    void (*funcs)(float, float [], int));
void svdvar(float **v, int ma, float w[], float **cvm);
void toeplz(float r[], float x[], float y[], int n);
void tptest(float data1[], float data2[], unsigned long n, float *t, float *prob);
void tqli(float d[], float e[], int n, float **z);
float trapzd(float (*func)(float), float a, float b, int n);
void tred2(float **a, int n, float d[], float e[]);
void tridag(float a[], float b[], float c[], float r[], float u[],
    unsigned long n);
float trncst(float x[], float y[], int iorder[], int ncity, int n[]);
void trnspt(int iorder[], int ncity, int n[]);
```

```
void ttest(float data1[], unsigned long n1, float data2[], unsigned long n2,
    float *t, float *prob);
void tutest(float data1[], unsigned long n1, float data2[], unsigned long n2,
    float *t, float *prob);
void twofft(float data1[], float data2[], float fft1[], float fft2[],
    unsigned long n);
void vander(double x[], double w[], double q[], int n);
void vegas(float regn[], int ndim, float (*fxn)(float [], float), int init,
    unsigned long ncall, int itmx, int nprn, float *tgral, float *sd,
    float *chi2a);
void voltra(int n, int m, float t0, float h, float *t, float **f,
    float (*g)(int, float), float (*ak)(int, int, float, float));
void wt1(float a[], unsigned long n, int isign,
    void (*wtstep)(float [], unsigned long, int));
void wtn(float a[], unsigned long nn[], int ndim, int isign,
    void (*wtstep)(float [], unsigned long, int));
void wwghts(float wghts[], int n, float h,
    void (*kermom)(double [], double ,int));
int zbrac(float (*func)(float), float *x1, float *x2);
void zbrak(float (*fx)(float), float x1, float x2, int n, float xb1[],
    float xb2[], int *nb);
float zbrent(float (*func)(float), float x1, float x2, float tol);
void zrhqr(float a[], int m, float rtr[], float rti[]);
float zriddr(float (*func)(float), float x1, float x2, float xacc);
void zroots(fcomplex a[], int m, fcomplex roots[], int polish);

#else /* ANSI */
/* traditional - K&R */

void addint();
void airy();
```

Rest of traditional declarations are here on the diskette.

```
#endif /* ANSI */

#endif /* _NR_H_ */
```

# 附录 B　实用例程



本书的许多算法都使用了下列的例程。第一个例程，nrerror，引用在当程序发生致命的错误时，返回适当的消息而中断程序。其它例程用来分配或释放矩阵或向量的内存，详见第1.2节。本书中，所有内存的分配和释放都运用这些例程作为媒介。因此，若想用不同方法分配内存，则只需改变这些例程，而无需更动程序本身。

这些例程在磁盘文件 nrutil.c 中。本附录列出了标题文件 nrutil.h 。在 nrutil.h 中，除了例程的说明外，还包含了适用于全书的一些宏指令。这些宏指令的思想已在第1.2 第中段述。

## B.1 内存分配：高级的议题

在此，将讨论两个关于向量和矩阵的内存分配问题。第一个问题是，我们的矩阵和其它 C 软件采用的矩阵之间：隐秘的(back-dool)"兼容性问题。第二个问题是，我们的单位偏移向量和矩阵的指针运算是否侵犯 ANSI C 的标准。

1、隐秘的兼容性。　正如第1.2 节中所阐述的，我们对于矩阵空间分配的方案经常采用"指向指针数组之行的指针"(详见图1.2.1)。通过这种方案，一旦矩阵的存储空间已分配，顶层的指针(矩阵的"名字")就可以用来在已分配的尺寸中，存储维数小或等于所分配维数尺寸的任意矩阵。只有当矩阵的存储空间最终释放后，所占用的空间才重新有意义。

基于这种方案，矩阵的每行并不需存储在连续的物理空间中，因为每行都具有指向自身的地址指针。但是，下列的内存分配程序，实际上(设计)是通过调用函数 malloc()，把一个完整的矩阵分配在一连续的块中，这种"隐秘的"事实就允许用户在其它软件中，直接采用这些例程来建立矩阵：矩阵 * *a 的第一个元素的地址(若 a 用如 a=matrix(1.m.1.n)的语句来分配，则它通常是 &a[1][1]；但若用零偏移量来建立，则它是 &a [0][0])保证是指向以行存储的全部物理矩阵的连续块的始端。这个地址可以作为一参数，被其它软件调用。(注意，在其它软件中，一般需要知道矩阵的物理空间的大小、或者矩阵的列数)。

2、单位偏移量。　在第1.2 节中，我们已经阐述了怎样通过指针运算 bb=b-1，从零偏移数组 [0..3] 获得单位偏移向量 bb[1..4]。因为指针 bb 指向 b 的前一位地址，所以 bb[1]的地址与 b [0]的地址相同，依此类推。

严格地说，ANSI C 标准并不赞同这种方案。问题是，事实上 b-1 指向未分配的空间；将永远不能引用存储单元 b-1 进行增量，而返回到已分配的空间中。当然，这个问题可能发生在少见的 b-1 完全无任何描述的情况中(只可能产生在分段式机器中)。若这种情况发生，则不能保证关系式 b=(b-1)-1 是满足的。

在实际中，人们并不用考虑这少见的问题。我们不认为在任何编译器和机器上，对于小整数 n，b=(b-n)+n 确实会失败。即使在分段式机器中，发生了编译器存储了某些 b-n 的表达式(也许是违反规则的)，那么，当这表达式加上 $n$ 后，b 就恢复正常。本书中的这些内存分配程序，从 1988年第一版发表后就已广泛应用，当然也存在有问障的问题，但是我们从未收到过在这方面失败的报告，那怕是一份简单的报告

(尽管有许多读者指出在理论上它可能失败，我们也已与C语言标准委员会通信，希望在将来[illegible]允准这个需要"b-(b-n)==n"(至少对n的某些范围，并设n为短型数据类型)。因为这样做，看来与[illegible]有的一些编辑器并没有什么冲突。

尽管没有任何实验上的(相对理论而言)问题，我们仍在本版中采取了许多措施，使我们的[illegible]的分配程序更好地与ANSI标准兼容。如下所列的，参量NR_END作为额外的存储单元数，它分配在每个向量块或矩阵块的起始端，以确保偏移指针的描述。我们设置NR_END的缺省值为1，它的效果是使所有单位偏移量的分配(例如，b=vector(1,7);或a=matrix(1,128,1,128);)严格地与ANSI标准兼容。当NR_END=1，存储单元浪费的数可忽略不计。在理论上，当偏移量大于1的分配仍是不兼容的(例如，b=vector(2,10))，而且在我们程序中，也不能得知它是否兼容。若需要进行这样的分配时，希望考虑增加NR-END的数值(注意，大的数值必然增加浪费的存储空间)。

以下是nrutil.h列表：

```
#ifndef _NR_UTILS_H_
#define _NR_UTILS_H_

static float sqrarg;
#define SQR(a) ((sqrarg=(a)) == 0.0 ? 0.0 : sqrarg*sqrarg)

static double dsqrarg;
#define DSQR(a) ((dsqrarg=(a)) == 0.0 ? 0.0 : dsqrarg*dsqrarg)

static double dmaxarg1,dmaxarg2;
#define DMAX(a,b) (dmaxarg1=(a),dmaxarg2=(b),(dmaxarg1) > (dmaxarg2) ?\
(dmaxarg1) : (dmaxarg2))

static double dminarg1,dminarg2;
#define DMIN(a,b) (dminarg1=(a),dminarg2=(b),(dminarg1) < (dminarg2) ?\
(dminarg1) : (dminarg2))

static float maxarg1,maxarg2;
#define FMAX(a,b) (maxarg1=(a),maxarg2=(b),(maxarg1) > (maxarg2) ?\
(maxarg1) : (maxarg2))

static float minarg1,minarg2;
#define FMIN(a,b) (minarg1=(a),minarg2=(b),(minarg1) < (minarg2) ?\
(minarg1) : (minarg2))

static long lmaxarg1,lmaxarg2;
#define LMAX(a,b) (lmaxarg1=(a),lmaxarg2=(b),(lmaxarg1) > (lmaxarg2) ?\
(lmaxarg1) : (lmaxarg2))

static long lminarg1,lminarg2;
#define LMIN(a,b) (lminarg1=(a),lminarg2=(b),(lminarg1) < (lminarg2) ?\
(lminarg1) : (lminarg2))

static int imaxarg1,imaxarg2;
#define IMAX(a,b) (imaxarg1=(a),imaxarg2=(b),(imaxarg1) > (imaxarg2) ?\
(imaxarg1) : (imaxarg2))

static int iminarg1,iminarg2;
#define IMIN(a,b) (iminarg1=(a),iminarg2=(b),(iminarg1) < (iminarg2) ?\
(iminarg1) : (iminarg2))

#define SIGN(a,b) ((b) >= 0.0 ? fabs(a) : -fabs(a))

#if defined(__STDC__) || defined(ANSI) || defined(NRANSI) /* ANSI */

void nrerror(char error_text[]);
float *vector(long nl, long nh);
int *ivector(long nl, long nh);
```

```
unsigned char *cvector(long nl, long nh);
unsigned long *lvector(long nl, long nh);
double *dvector(long nl, long nh);
float **matrix(long nrl, long nrh, long ncl, long nch);
double **dmatrix(long nrl, long nrh, long ncl, long nch);
int **imatrix(long nrl, long nrh, long ncl, long nch);
float **submatrix(float **a, long oldrl, long oldrh, long oldcl, long oldch,
    long newrl, long newcl);
float **convert_matrix(float *a, long nrl, long nrh, long ncl, long nch);
float ***f3tensor(long nrl, long nrh, long ncl, long nch, long ndl, long ndh);
void free_vector(float *v, long nl, long nh);
void free_ivector(int *v, long nl, long nh);
void free_cvector(unsigned char *v, long nl, long nh);
void free_lvector(unsigned long *v, long nl, long nh);
void free_dvector(double *v, long nl, long nh);
void free_matrix(float **m, long nrl, long nrh, long ncl, long nch);
void free_dmatrix(double **m, long nrl, long nrh, long ncl, long nch);
void free_imatrix(int **m, long nrl, long nrh, long ncl, long nch);
void free_submatrix(float **b, long nrl, long nrh, long ncl, long nch);
void free_convert_matrix(float **b, long nrl, long nrh, long ncl, long nch);
void free_f3tensor(float ***t, long nrl, long nrh, long ncl, long nch,
    long ndl, long ndh);

#else /* ANSI */
/* traditional - K&R */

void nrerror();
float *vector();
Rest of traditional declarations are here on the diskette.

#endif /* ANSI */

#endif /* _NR_UTILS_H_ */
```

以下是 nrutil.c 文件：

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#define NR_END 1
#define FREE_ARG char*

void nrerror(char error_text[])
/* Numerical Recipes standard error handler */
{
     fprintf(stderr,"Numerical Recipes run-time error...\n");
     fprintf(stderr,"%s\n",error_text);
     fprintf(stderr,"...now exiting to system...\n");
     exit(1);
}

float *vector(long nl, long nh)
/* allocate a float vector with subscript range v[nl..nh] */
{
    float *v;

    v=(float *)malloc((size_t) ((nh-nl+1+NR_END)*sizeof(float)));
    if (!v) nrerror("allocation failure in vector()");
    return v-nl+NR_END;
}

int *ivector(long nl, long nh)
/* allocate an int vector with subscript range v[nl..nh] */
{
    int *v;
```

```
	v=(int *)malloc((size_t) ((nh-nl+1+NR_END)*sizeof(int)));
	if (!v) nrerror("allocation failure in ivector()");
	return v-nl+NR_END;
}

unsigned char *cvector(long nl, long nh)
/* allocate an unsigned char vector with subscript range v[nl..nh] */
{
	unsigned char *v;

	v=(unsigned char *)malloc((size_t) ((nh-nl+1+NR_END)*sizeof(unsigned char)));
	if (!v) nrerror("allocation failure in cvector()");
	return v-nl+NR_END;
}

unsigned long *lvector(long nl, long nh)
/* allocate an unsigned long vector with subscript range v[nl..nh] */
{
	unsigned long *v;

	v=(unsigned long *)malloc((size_t) ((nh-nl+1+NR_END)*sizeof(long)));
	if (!v) nrerror("allocation failure in lvector()");
	return v-nl+NR_END;
}

double *dvector(long nl, long nh)
/* allocate a double vector with subscript range v[nl..nh] */
{
	double *v;

	v=(double *)malloc((size_t) ((nh-nl+1+NR_END)*sizeof(double)));
	if (!v) nrerror("allocation failure in dvector()");
	return v-nl+NR_END;
}

float **matrix(long nrl, long nrh, long ncl, long nch)
/* allocate a float matrix with subscript range m[nrl..nrh][ncl..nch] */
{
	long i, nrow=nrh-nrl+1,ncol=nch-ncl+1;
	float **m;

	/* allocate pointers to rows */
	m=(float **) malloc((size_t)((nrow+NR_END)*sizeof(float*)));
	if (!m) nrerror("allocation failure 1 in matrix()");
	m += NR_END;
	m -= nrl;

	/* allocate rows and set pointers to them */
	m[nrl]=(float *) malloc((size_t)((nrow*ncol+NR_END)*sizeof(float)));
	if (!m[nrl]) nrerror("allocation failure 2 in matrix()");
	m[nrl] += NR_END;
	m[nrl] -= ncl;

	for(i=nrl+1;i<=nrh;i++) m[i]=m[i-1]+ncol;

	/* return pointer to array of pointers to rows */
	return m;
}

double **dmatrix(long nrl, long nrh, long ncl, long nch)
/* allocate a double matrix with subscript range m[nrl..nrh][ncl..nch] */
{
	long i, nrow=nrh-nrl+1,ncol=nch-ncl+1;
	double **m;

	/* allocate pointers to rows */
	m=(double **) malloc((size_t)((nrow+NR_END)*sizeof(double*)));
```

```
    if (!m) nrerror("allocation failure 1 in matrix()");
    m += NR_END;
    m -= nrl;

    /* allocate rows and set pointers to them */
    m[nrl]=(double *) malloc((size_t)((nrow*ncol+NR_END)*sizeof(double)));
    if (!m[nrl]) nrerror("allocation failure 2 in matrix()");
    m[nrl] += NR_END;
    m[nrl] -= ncl;

    for(i=nrl+1;i<=nrh;i++) m[i]=m[i-1]+ncol;

    /* return pointer to array of pointers to rows */
    return m;
}

int **imatrix(long nrl, long nrh, long ncl, long nch)
/* allocate a int matrix with subscript range m[nrl..nrh][ncl..nch] */
{
    long i, nrow=nrh-nrl+1,ncol=nch-ncl+1;
    int **m;

    /* allocate pointers to rows */
    m=(int **) malloc((size_t)((nrow+NR_END)*sizeof(int*)));
    if (!m) nrerror("allocation failure 1 in matrix()");
    m += NR_END;
    m -= nrl;


    /* allocate rows and set pointers to them */
    m[nrl]=(int *) malloc((size_t)((nrow*ncol+NR_END)*sizeof(int)));
    if (!m[nrl]) nrerror("allocation failure 2 in matrix()");
    m[nrl] += NR_END;
    m[nrl] -= ncl;

    for(i=nrl+1;i<=nrh;i++) m[i]=m[i-1]+ncol;

    /* return pointer to array of pointers to rows */
    return m;
}
float **submatrix(float **a, long oldrl, long oldrh, long oldcl, long oldch,
    long newrl, long newcl)
/* point a submatrix [newrl..][newcl..] to a[oldrl..oldrh][oldcl..oldch] */
{
    long i,j,nrow=oldrh-oldrl+1,ncol=oldcl-newcl;
    float **m;

    /* allocate array of pointers to rows */
    m=(float **) malloc((size_t) ((nrow+NR_END)*sizeof(float*)));
    if (!m) nrerror("allocation failure in submatrix()");
    m += NR_END;
    m -= newrl;

    /* set pointers to rows */
    for(i=oldrl,j=newrl;i<=oldrh;i++,j++) m[j]=a[i]+ncol;

    /* return pointer to array of pointers to rows */
    return m;
}

float **convert_matrix(float *a, long nrl, long nrh, long ncl, long nch)
/* allocate a float matrix m[nrl..nrh][ncl..nch] that points to the matrix
declared in the standard C manner as a[nrow][ncol], where nrow=nrh-nrl+1
and ncol=nch-ncl+1. The routine should be called with the address
&a[0][0] as the first argument. */
{
    long i,j,nrow=nrh-nrl+1,ncol=nch-ncl+1;
```

```
    float **m;

    /* allocate pointers to rows */
    m=(float **) malloc((size_t) ((nrow+NR_END)*sizeof(float*)));
    if (!m)    nrerror("allocation failure in convert_matrix()");
    m += NR_END;
    m -= nrl;

    /* set pointers to rows */
    m[nrl]=a-ncl;
    for(i=1,j=nrl+1;i<nrow;i++,j++) m[j]=m[j-1]+ncol;
    /* return pointer to array of pointers to rows */
    return m;
}

float ***f3tensor(long nrl, long nrh, long ncl, long nch, long ndl, long ndh)
/* allocate a float 3tensor with range t[nrl..nrh][ncl..nch][ndl..ndh] */
{
    long i,j,nrow=nrh-nrl+1,ncol=nch-ncl+1,ndep=ndh-ndl+1;
    float ***t;

    /* allocate pointers to pointers to rows */
    t=(float ***) malloc((size_t)((nrow+NR_END)*sizeof(float**)));
    if (!t) nrerror("allocation failure 1 in f3tensor()");
    t += NR_END;
    t -= nrl;

    /* allocate pointers to rows and set pointers to them */
    t[nrl]=(float **) malloc((size_t)((nrow*ncol+NR_END)*sizeof(float*)));
    if (!t[nrl]) nrerror("allocation failure 2 in f3tensor()");
    t[nrl] += NR_END;
    t[nrl] -= ncl;

    /* allocate rows and set pointers to them */
    t[nrl][ncl]=(float *) malloc((size_t)((nrow*ncol*ndep+NR_END)*sizeof(float)));
    if (!t[nrl][ncl]) nrerror("allocation failure 3 in f3tensor()");
    t[nrl][ncl] += NR_END;
    t[nrl][ncl] -= ndl;

    for(j=ncl+1;j<=nch;j++) t[nrl][j]=t[nrl][j-1]+ndep;
    for(i=nrl+1;i<=nrh;i++) {
        t[i]=t[i-1]+ncol;
        t[i][ncl]=t[i-1][ncl]+ncol*ndep;
        for(j=ncl+1;j<=nch;j++) t[i][j]=t[i][j-1]+ndep;
    }

    /* return pointer to array of pointers to rows */
    return t;
}

void free_vector(float *v, long nl, long nh)
/* free a float vector allocated with vector() */
{
    free((FREE_ARG) (v+nl-NR_END));
}

void free_ivector(int *v, long nl, long nh)
/* free an int vector allocated with ivector() */
{
    free((FREE_ARG) (v+nl-NR_END));
}

void free_cvector(unsigned char *v, long nl, long nh)
/* free an unsigned char vector allocated with cvector() */
{
    free((FREE_ARG) (v+nl-NR_END));
}
```

```
void free_lvector(unsigned long *v, long nl, long nh)
/* free an unsigned long vector allocated with lvector() */
{
    free((FREE_ARG) (v+nl-NR_END));
}

void free_dvector(double *v, long nl, long nh)
/* free a double vector allocated with dvector() */
{
    free((FREE_ARG) (v+nl-NR_END));
}

void free_matrix(float **m, long nrl, long nrh, long ncl, long nch)
/* free a float matrix allocated by matrix() */
{
    free((FREE_ARG) (m[nrl]+ncl-NR_END));
    free((FREE_ARG) (m+nrl-NR_END));
}

void free_dmatrix(double **m, long nrl, long nrh, long ncl, long nch)
/* free a double matrix allocated by dmatrix() */
{
    free((FREE_ARG) (m[nrl]+ncl-NR_END));
    free((FREE_ARG) (m+nrl-NR_END));
}

void free_imatrix(int **m, long nrl, long nrh, long ncl, long nch)
/* free an int matrix allocated by imatrix() */
{
    free((FREE_ARG) (m[nrl]+ncl-NR_END));
    free((FREE_ARG) (m+nrl-NR_END));
}
void free_submatrix(float **b, long nrl, long nrh, long ncl, long nch)
/* free a submatrix allocated by submatrix() */
{
    free((FREE_ARG) (b+nrl-NR_END));
}

void free_convert_matrix(float **b, long nrl, long nrh, long ncl, long nch)
/* free a matrix allocated by convert_matrix() */
{
    free((FREE_ARG) (b+nrl-NR_END));
}

void free_f3tensor(float ***t, long nrl, long nrh, long ncl, long nch,
    long ndl, long ndh)
/* free a float f3tensor allocated by f3tensor() */
{
    free((FREE_ARG) (t[nrl][ncl]+ndl-NR_END));
    free((FREE_ARG) (t[nrl]+ncl-NR_END));
    free((FREE_ARG) (t+nrl-NR_END));
}
```

# 附录 C　复数运算

下列函数作为复数运算用于程序 **cisi**、**frenel**、**hypdrv**、**hypgeo**、**hypser**、**laguer**、**zroots** 和 **fixrts** 中。一个复数定义成包含二个浮点型数值的结构，一个为实部(.r)，一个为虚部(.i)。复数变量以数值进行传送和返回。其它讨论见第 1.2 节。

本附录在磁盘的文件 complex.c 中

```c
#include <math.h>

typedef struct FCOMPLEX {float r,i;} fcomplex;

fcomplex Cadd(fcomplex a, fcomplex b)
{
    fcomplex c;
    c.r=a.r+b.r;
    c.i=a.i+b.i;
    return c;
}

fcomplex Csub(fcomplex a, fcomplex b)
{
    fcomplex c;
    c.r=a.r-b.r;
    c.i=a.i-b.i;
    return c;
}


fcomplex Cmul(fcomplex a, fcomplex b)
{
    fcomplex c;
    c.r=a.r*b.r-a.i*b.i;
    c.i=a.i*b.r+a.r*b.i;
    return c;
}

fcomplex Complex(float re, float im)
{
    fcomplex c;
    c.r=re;
    c.i=im;
    return c;
}

fcomplex Conjg(fcomplex z)
{
    fcomplex c;
    c.r=z.r;
    c.i = -z.i;
    return c;
}

fcomplex Cdiv(fcomplex a, fcomplex b)
{
    fcomplex c;
    float r,den;
    if (fabs(b.r) >= fabs(b.i)) {
        r=b.i/b.r;
```

```
        den=b.r+r*b.i;
        c.r=(a.r+r*a.i)/den;
        c.i=(a.i-r*a.r)/den;
    } else {
        r=b.r/b.i;
        den=b.i+r*b.r;
        c.r=(a.r*r+a.i)/den;
        c.i=(a.i*r-a.r)/den;
    }
    return c;
}

float Cabs(fcomplex z)
{
    float x,y,ans,temp;
    x=fabs(z.r);
    y=fabs(z.i);
    if (x == 0.0)
        ans=y;
    else if (y == 0.0)
        ans=x;
    else if (x > y) {
        temp=y/x;
        ans=x*sqrt(1.0+temp*temp);
    } else {
        temp=x/y;
        ans=y*sqrt(1.0+temp*temp);
    }
    return ans;
}

fcomplex Csqrt(fcomplex z)
{
    fcomplex c;
    float x,y,w,r;
    if ((z.r == 0.0) && (z.i == 0.0)) {
        c.r=0.0;
        c.i=0.0;
        return c;
    } else {
        x=fabs(z.r);
        y=fabs(z.i);
        if (x >= y) {
            r=y/x;
            w=sqrt(x)*sqrt(0.5*(1.0+sqrt(1.0+r*r)));
        } else {
            r=x/y;
            w=sqrt(y)*sqrt(0.5*(r+sqrt(1.0+r*r)));
        }
        if (z.r >= 0.0) {
            c.r=w;
            c.i=z.i/(2.0*w);
        } else {
            c.i=(z.i >= 0) ? w : -w;
            c.r=z.i/(2.0*c.i);
        }
        return c;
    }
}

fcomplex RCmul(float x, fcomplex a)
{
    fcomplex c;
    c.r=x*a.r;
    c.i=x*a.i;
    return c;
}
```

# 程序从属表

下面，按字母顺序列出《C语言数值算法程序大全》一书中所有的程序。当某程序需用到其它子程序时，无论是从本书中取得或用户提供，其依从关系用从属树图表示：一个程序直接调用的所有程序列在紧靠它的右边一列中，用直实线连接；间接调用的程序列在右边其它列中，用直实线将它与相关程序连接起来。印刷符号的约定为：书中的程序用打印体形式（如 eulsum 和 *gammln*）。而斜字体的程序用来表示单个从属树中其次和随后发生的程序（当从磁盘中获取程序或者它们的归档文件时，只需用正字体）。用户需提供的程序用正文体和方括号指出，例如[funcv]。为了便于查阅这些程序的正文说明，列表最右端列出了每个程序所处的章节。

```
addint ── interp ........................................ §19.6
airy ──┬── bessik ─┐ .................................... §6.7
       └── bessjy ─┴── beschb ── chebev
amebsa ─┬── ran1 ........................................ §10.9
        ├── amotsa ──┬── [funk]
        │            └── ran1
        └── [funk]
amoeba ─┬── amotry ── [funk] ............................ §10.4
        └── [funk]
amotry ── [funk] ........................................ §10.4
amotsa ─┬── [funk] ...................................... §10.9
        └── ran1
anneal ─┬── ran3 ........................................ §10.9
        ├── irbit1
        ├── trncst
        ├── metrop ── ran3
        ├── trnspt
        ├── revcst
        └── reverse
anorm2 .................................................. §19.6
arcmak .................................................. §20.5
arcode ── arcsum ........................................ §20.5
arcsum .................................................. §20.5
avevar .................................................. §14.2
badluk ─┬── julday ...................................... §1.1
        └── flmoon
balanc .................................................. §11.5
banbks .................................................. §2.4
bandec .................................................. §2.4
banmul .................................................. §2.4
```

bcucof ........ § 3.6

bcuint —— bcucof ........ § 3.6

beschb —— chebev ........ § 6.7

bessi —— bessi0 ........ § 6.6

bessi0 ........ § 6.6

bessi1 ........ § 6.6

bessik —— beschb —— chebev ........ § 6.7

bessj ┬ bessj0 ........ § 6.5
└ bessj1

bessj0 ........ § 6.5

bessj1 ........ § 6.5

bessjy —— beschb —— chebev ........ § 6.7

bessk ┬ bessk0 —— bessi0 ........ § 6.6
└ bessk1 —— bessi1

bessk0 —— bessi0 ........ § 6.6

bessk1 —— bessi1 ........ § 6.6

bessy ┬ bessy1 —— bessj1 ........ § 6.5
└ bessy0 —— bessj0

bessy0 —— bessj0 ........ § 6.5

bessy1 —— bessj1 ........ § 6.5

beta —— gammln ........ § 6.1

betacf ........ § 6.4

betai ┬ gammln ........ § 6.4
└ betacf

bico —— factln —— gammln ........ § 6.1

bksub ........ § 17.3

bnldev ┬ ran1 ........ § 7.3
└ gammln

brent —— [func] ........ § 10.2

broydn ┬ fmin ┐ ........ § 9.7
├ fdjac ┴ [funcv]
├ qrdcmp
├ qrupdt —— rotate
├ rsolv
└ lnsrch —— *fmin* —— [funcv]

bsstep ┬ mmid —— [derivs] ........ § 16.4
└ pzextr

caldat ........ § 1.1

chder ........ § 5.9

chebev ........ § 5.8

chebft —— [func] ........ § 5.8

chebpc ........ § 5.10

chint ........ § 5.9

chixy …… § 15.5
choldc …… § 2.9
cholsl …… § 2.9
chsone — gammq — gser — gammln …… § 14.3
— gcf — gammln
chstwo — gammq — gser — …… § 14.3
— gcf — gammln
cisi …… § 6.9
cntab1 — gammq — gser — …… § 14.4
— gcf — gammln
cntab2 …… § 14.4
convlv — twofft — four1 …… § 13.1
— realft — four1
copy …… § 1[illegible].6
correl — twofft — four1 …… § 13.2
— realft — four1
cosft1 realft — four1 …… § 12.3
cosft2 — realft — four1 …… § 12.3
covsrt …… § 15.4
crank …… § 14.6
cyclic — tridag …… § 2.7
daub4 …… § 13.10
dawson …… § 6.10
dbrent — [func] …… § 1[illegible]
— [dfunc]
ddpoly …… § 5.3
decchk …… § 20.3
df1dim — [dfunc] …… § 10.6
dfour1 …… four1 的双精度版，见该项
dfpmin — [func] …… § 10.7
— [dfunc]
— lnsrch — [func]
dfridr — [func] …… § 5.7
dftcor …… § 13.9
dftint — [func] …… § 13.9
— realft — four1
— polint
— dftcor
dfleq …… § 17.4
dlinmin — mnbrak — f1dim — [func] …… § 10.6
— dbrent — df1dim — [func]
dpythag …… pythag 的双精度版，见该项
drealft …… realft 的双精度版，见该项
dsprsax …… sprsax 的双精度版，见该项

dsprstx …… sprstx 的双精度版，见该项

dsvbksh …… svbksb 的双精度版，见该项

dsvdcmp …… svdcmp 的双精度版，见该项

eclass …… § 8.6

eclazz …… § 8.6

ei …… § 6.3

eigsrt …… § 11.1

elle ─┬─ rf …… § 6.11
      └─ rd

ellf ── rf …… § 6.11

ellpi ─┬─ rf …… § 6.11
       └─ rj ─┬─ rc
              └─ *rf*

elmhes …… § 11.5

erff ── gammp ─┬─ gser ┐ …… § 6.2
               └─ gcf ─┴─ gammln

erffc ─┬─ gammp ─┬─ gser ┐ …… § 6.2
       │         └─ gcf ─┴─ gammln
       └─ gammq ─┬─ *gser* ┐
                 └─ *gcf* ─┴─ *gammln*

erfcc …… § 6.2

eulsum …… § 5.1

evlmem …… § 13.7

expdev ── ran1 …… § 7.2

expint …… § 6.3

f1dim ── [func] …… § 10.5

factln ── gammln …… § 6.1

factrl ── gammln …… § 6.1

fasper ─┬─ avevar …… § 13.8
        ├─ spread
        └─ realft ── four1

fdjac ── [funcv] …… § 9.7

fgauss …… § 15.5

fill0 …… § 19.6

fit ── gammq ─┬─ gser ┐ …… § 15.2
              └─ gcf ─┴─ gammln

fitexy ─┬─ avevar …… § 15.3
        ├─ fit ── gammq ─┬─ gser ┐
        │                └─ gcf ─┴─ gammln
        ├─ chixy
        ├─ mnbrak
        ├─ brent
        ├─ *gammq* ─┬─ *gser* ┐
        │           └─ *gcf* ─┴─ *gammln*
        └─ zbrent ── *chixy*

fixrts — zroots — laguer ······ § 13.6

fleg ······ § 15.4

flmoon ······ § 1.0

fmin — [funcv] ······ § 9.7

four1 ······ § 12.2

fourew ······ § 12.6

fourfs — fourew ······ § 12.6

fourn ······ § 12.4

fpoly ······ § 15.4

fred2 ┬ gauleg ······ § 18.1
  ├ [ak]
  ├ [g]
  ├ ludcmp
  └ lubksb

fredex ┬ quadmx — wwghts — kermom ······ § 18.3
  ├ ludcmp
  └ lubksb

fredin ┬ [ak] ······ § 18.1
  └ [g]

frenel ······ § 6.9

frprmn ┬ [func] ······ § 10.6
  ├ [dfunc]
  └ linmin ┬ mnbrak ┐
    └ brent ┘ — f1dim — [func]

ftest ┬ avevar ······ § 14.2
  └ betai ┬ gammln
    └ betacf

gamdev — ran1 ······ § 7.3

gammln ······ § 6.1

gammp ┬ gser ┐ ······ § 6.2
  └ gcf ┘ — gammln

gammq ┬ gser ┐ ······ § 6.2
  └ gcf ┘ — gammln

gasdev — ran1 ······ § 7.2

gaucof ┬ tqli — pythag ······ § 4.5
  └ eigsrt

gauher ······ § 4.5

gaujac — gammln ······ § 4.5

gaulag — gammln ······ § 4.5

gauleg ······ § 4.5

garssj ······ § 2.1

gcf — gammln ······ § 6.2

golden — [func] ······ § 10.1

gser — gammln ······ § 6.2

hpsel — sort ······ § 8.5

hpsort ······ § 8.3
hqr ······ § 11.6
hufapp ······ § 20.4
hufdec ······ § 20.4
hufenc ······ § 20.4
hufmak —— hufapp ······ § 20.4
hunt ······ § 3.4
hypdrv ······ § 6.12
hypgeo ┬ hypser ······ § 6.12
  └ odeint ┬ bsstep ┬ mmid
      │   └ pzextr
      └ hypdrv
hypser ······ § 6.12
icrc —— icrc1 ······ § 20.3
icrc1 ······ § 20.3
igray ······ § 20.2
iindexx ······ indexx 的长整型版，见该项
indexx ······ § 8.4
interp ······ § 19.6
irbit1 ······ § 7.4
irbit2 ······ § 7.4
jacobi ······ § 11.1
jacobn ······ § 16.6
julday ······ § 1.1
kendl1 —— erfcc ······ § 14.6
kendl2 —— erfcc ······ § 14.6
kermom ······ § 18.3
ks2d1s ┬ quadct ······ § 14.7
  ├ quadvl
  ├ pearsn —— betai ┬ gammln
  │       └ betacf
  └ probks
ks2d2s ┬ quadct ······ § 14.7
  ├ pearsn —— betai ┬ gammln
  │       └ betacf
  └ probks
ksone ┬ sort ······ § 14.3
  ├ [func]
  └ probks
kstwo ┬ sort ······ § 14.3
  └ probks
laguer ······ § 9.5

lfit —— [funcs] ········ § 15.4
├— gaussj
└— covsrt
linbcg ┬— atimes ········ § 2.7
├— snrm
└— asolve
linmin ┬— mnbrak ┐ ········ § 10.5
└— brent ——┴ f1dim —— [func]
lnsrch —— [func] ········ § 9.7
locate ········ § 3.4
lop ········ § 19.6
lubksb ········ § 2.3
ludcmp ········ § 2.3
machar ········ § 20.1
matadd ········ § 19.6
matsub ········ § 19.6
medfit —— rofunc —— select ········ § 15.7
memcof ········ § 13.6
metrop —— ran3 ········ § 10.9
mgfas ┬— rstrct ········ § 19.6
├— slvsm2 —— fill0
├— interp
├— copy
├— relax2
├— lop
├— matsub
├— anorm2
└— matadd
mglin —— rstrct ········ § 19.6
├— slvsml —— fill0
├— interp
├— copy
├— relax
├— resid
├— *fill0*
└— addint —— *interp*
midinf —— [func] ········ § 4.4
midpnt —— [func] ········ § 4.4
miser ┬— ranpt —— ran1 ········ § 7.8
└— [func]
mmid —— [derivs] ········ § 16.3
mnbrak —— [func] ········ § 10.1
mnewt —— [usrfun] ········ § 9.6
├— ludcmp
└— lubksb
moment ········ § 11.1

mp2dfr — mpops …… § 20.6

mpdiv ┬ mpinv ┬ mpmul — drealft — dfour1 …… § 20.6
│ └ mpops
├ *mpmul* — *drealft* — *dfour1*
└ mpops

mpinv ┬ mpmul — drealft — dfour1 …… § 20.6
└ mpops

mpmul — drealft — dfour1 …… § 20.6

mpops …… § 20.6

mppi ┬ mpsqrt ┬ mpmul — drealft — dfour1 …… § 20.6
│ └ mpops
├ *mpops*
├ *mpmul* — *drealft* — *dfour1*
├ mpinv — *mpmul* — *drealft* — *dfour1*
└ mp2dfr — *mpops*

mprove — lubksb …… § 2.5

mpsqrt ┬ mpmul — drealft — dfour1 …… § 20.6
└ mpops

mrqcof — [funcs] …… § 15.5

mrqmin ┬ mrqcof …… § 15.5
├ gaussj
└ covsrt

newt ┬ fmin ┐ …… § 9.7
├ fdjac ┘ — [funcv]
├ ludcmp
├ lubksb
└ lnsrch — *fmin* — [funcv]

odeint ┬ [derivs] …… § 16.2
└ rkqs ┬ [derivs]
└ rkck — [derivs]

orthog …… 4.5

pade ┬ ludcmp …… § 5.12
├ lubksb
└ mprove — *lubksb*

pccheb …… § 5.11

pcshft …… § 5.10

pearsn — betai ┬ gammln …… § 14.5
└ betacf

period — avevar …… § 13.8

piksr2 …… § 8.1

piksrt …… § 8.1

pinvs …… § 17.3

plgndr …… § 6.8

poidev ┬ ran1 …… § 7.3
└ gammln

polcoe …… § 3.5

polcof — polint …… § 3.5

poldiv ........ §5.3
polin2 — polint ........ §3.6
polint ........ §3.1
powell ┬─ [func] ........ §10.5
└─ linmin ┬─ mnbrak ┐
└─ brent ┴─ f1dim ── [func]
predic ........ §13.6
probks ........ §14.3
psdes ........ §7.5
pwt ........ §13.10
pwtset ........ §13.10
pythag ........ §2.6
pzextr ........ §16.4
qgaus ── [func] ........ §4.5
qrdcmp ........ §2.10
qromb ┬─ trapzd ── [func] ........ §4.3
└─ polint
qromo ┬─ midpnt ── [func] ........ §4.4
└─ polint
qroot ── poldiv ........ §9.5
qrsolv ── rsolv ........ §2.10
qrupdt ── rotate ........ §2.10
qsimp ── trapzd ── [func] ........ §4.2
qtrap ── trapzd ── [func] ........ §4.2
quad3d ── qgaus ┬─ [func] ........ §4.6
├─ [y1]
├─ [y2]
├─ [z1]
└─ [z2]
quadct ........ §14.7
quadmx ── wwghts ── kermom ........ §18.3
quadvl ........ §14.7
ran0 ........ §7.1
ran1 ........ §7.1
ran2 ........ §7.1
ran3 ........ §7.1
ran4 ── psdes ........ §7.5
rank ........ §8.4
ranpt ── ran1 ........ §7.8
ratint ........ §3.2

ratlsq — [fn] ⋯⋯ § 5.13
  ├ dsvdcmp — dpythag
  ├ dsvbksb
  └ ratval
ratval ⋯⋯ § 5.3
rc ⋯⋯ § 6.11
rd ⋯⋯ § 6.11
realft — four1 ⋯⋯ § 12.3
rebin ⋯⋯ § 7.8
red ⋯⋯ § 17.3
relax ⋯⋯ § 19.6
relax2 ⋯⋯ § 19.6
resid ⋯⋯ § 19.6
revcst ⋯⋯ § 10.9
reverse ⋯⋯ § 10.9
rf ⋯⋯ § 6.11
rj ┬ rc ⋯⋯ § 6.11
  └ rf
rk4 — [derivs] ⋯⋯ § 16.1
rkck — [derivs] ⋯⋯ § 16.2
rkdumb ┬ [derivs] ⋯⋯ § 16.1
  └ rk4 — [derivs]
rkqs — rkck — [derivs] ⋯⋯ § 16.2
rlft3 — fourn ⋯⋯ § 12.5
rofunc — select ⋯⋯ § 15.7
rotate ⋯⋯ § 2.10
rsolv ⋯⋯ § 2.10
rstrct ⋯⋯ § 19.6
rtbis — [func] ⋯⋯ § 9.1
rtflsp — [func] ⋯⋯ § 9.2
rtnewt — [funcd] ⋯⋯ § 9.4
rtsafe — [funcd] ⋯⋯ § 9.4
rtsec — [func] ⋯⋯ § 9.2
rzextr ⋯⋯ § 16.4
savgol ┬ ludcmp ⋯⋯ § 14.8
  └ lubksb
scrsho — func ⋯⋯ § 9.0
select ⋯⋯ § 8.5
selip — shell ⋯⋯ § 8.5

sfroid —— plgndr ………… § 17.4
— solvde ┬ difeq
├ pinvs
├ red
└ bksub

shell ………… § 8.1

shoot ┬ [load] ………… § 17.1
├ odeint ┬ [derivs]
│ └ rkqs —— rkck — [derivs]
└ [score]

shootf ┬ [load1] ………… § 17.2
├ odeint ┬ [derivs]
│ └ rkqs —— rkck —— [derivs]
├ [score]
└ [load2]

simp1 ………… § 10.8

simp2 ………… § 10.8

simp3 ………… § 10.8

simplx ┬ simp1 ………… § 10.8
├ simp2
└ simp3

simpr ┬ ludcmp ………… § 16.6
├ lubksb
└ [derivs]

sinft —— realft —— four1 ………… § 12.3

slvsm2 —— fill0 ………… § 19.6

slvsml —— fill0 ………… § 19.6

sncndn ………… § 6.11

snrm ………… § 2.7

sobseq ………… § 7.7

solvde ┬ difeq ………… § 17.3
├ pinvs
├ red
└ bksub

sor ………… § 19.5

sort ………… § 8.2

sort2 ………… § 8.2

sort3 —— indexx ………… § 8.4

spctrm —— four1 ………… § 13.4

spear ┬ sort2 ………… § 14.6
├ crank
├ erfcc
└ betai ┬ gammln
└ betacf

sphbes —— bessjy —— beschb — chebev ………… § 6.7

```
sphfpt       newt ─┬─ fdjac ──── shootf(q,v) ································ § 17.4
                   ├─ lnsrch ─┐
                   ├─ fmin ───┴── shootf(q,v)
                   ├─ ludcmp
                   └─ lubksb
sphoot     ── newt ─┬─ fdjac ──── shoot(q,v) ································ § 17.4
                    ├─ lnsrch ─┐
                    ├─ fmin    │  shoot(q,v)
                    ├─ ludcmp0
                    └─ lubksb
splie2 ──── spline ··························································· § 3.6
splin2 ──┬─ splint ··························································· § 3.6
         └─ splint
spline ······································································· § 3.3
splint ······································································· § 3.3
spread ······································································· § 13.8
sprsax ······································································· § 2.7
sprsin ······································································· § 2.7
sprspm ······································································· § 2.7
sprstm ······································································· § 2.7
sprstp ──── iindexx ·························································· § 2.7
sprstx ······································································· § 2.7
stifbs ──┬─ jacobn ··························································· § 16.6
         ├─ simpr ──┬─ ludcmp
         │          └─ luksb
         └─ pzextr
stiff ──┬─ jacobn ···························································· § 16.6
        ├─ ludcmp
        ├─ lubksb
        └─ [derivs]
stoerm       [derivs] ························································ § 16.5
svbksb ······································································· § 2.6
svdcmp ──── pythag ··························································· § 2.6
svdfit ──┬─ [funcs] ·························································· § 15.4
         ├─ svdcmp      pythag
         └─ svbksb
svdvar ······································································· § 15.4
toeplz ······································································· § 2.8
tptest ──┬─ avevar ··························································· § 14.2
         └─ betai ──┬─ gammln
                    └─ betacf
tqli ──── pythag ····························································· § 11.3
trapzd ──── [func] ··························································· § 4.2
tred2 ········································································ § 11.2
```

tridag ........ § 2.4

trnest ........ § 10.9

trnspt ........ § 10.9

ttest ─┬─ avevar ........ § 14.2
└─ betai ─┬─ gammln
└─ betacf

tutest ─┬─ avevar ........ § 14.2
└─ betai ─┬─ gammln
└─ betacf

twofft ─── four1 ........ § 12.3

vander ........ § 2.8

vegas ─┬─ rebin ........ § 7.8
├─ ran2
└─ [fxn]

voltra ─┬─ [g] ........ § 18.2
├─ [ak]
├─ ludcmp
└─ lubksb

wt1 ─── daub4 ........ § 13.10

wtn ─── daub4 ........ § 13.10

wwghts ─── kermom ........ § 18.3

zbrac ─── [func] ........ § 9.1

zbrak ─── [func] ........ § 9.1

zbrent ─── [func] ........ § 9.3

zrhqr ─┬─ balanc ........ § 9.5
└─ hqr

zriddr ─── [func] ........ § 9.2

zroots ─── laguer ........ § 9.5

# 各章节的计算机程序

| | | |
|---|---|---|
| 1.0 | flmoon | 按日期计算月相 |
| 1.1 | julday | 从历书日期计算凯撒历日数 |
| 1.1 | badluk | 13日正好星期五,并又为满月 |
| 1.1 | caldat | 从凯撒历日数计算历书日期 |
| 2.1 | gaussj | 矩阵求逆和线性方程求解-高斯-约当法 |
| 2.3 | ludcmp | 线性方程求解,$LU$ 分解 |
| 2.3 | lubksb | 线性方程求解,回代过程 |
| 2.4 | tridag | 三对角线性方程求解 |
| 2.4 | banmul | 与带状对角矩阵乘得的向量 |
| 2.4 | bandec | 带状对角方程求解,分解法 |
| 2.4 | banbks | 带状对角方程求解,回代过程 |
| 2.5 | mprove | 线性方程求解,迭代改进 |
| 2.6 | svbksb | 奇异值回代过程 |
| 2.6 | svdcmp | 矩阵奇异值分解 |
| 2.6 | pythag | 计算$(a^2+b^2)^{1/2}$,不发生溢出 |
| 2.7 | cyclic | 周期三对角方程求解 |
| 2.7 | sprsin | 把矩阵转换成稀疏方式 |
| 2.7 | sprsax | 稀疏矩阵和向量的乘积 |
| 2.7 | sprstx | 稀疏矩阵转置和向量的乘积 |
| 2.7 | sprstp | 稀疏矩阵的转置 |
| 2.7 | sprspm | 两个稀疏矩阵的模式乘 |
| 2.7 | sprstm | 两个稀疏矩阵的阈值乘 |
| 2.7 | linbcg | 稀疏线性方程的共轭梯度法求解 |
| 2.7 | snrm | 用于 **linbcg** 的计算向量模 |
| 2.7 | atimes | 用于 **linbcg**,计算稀疏矩阵或其转置与向量的乘积 |
| 2.7 | asolve | 用于 **linbcg** 的先决条件 |
| 2.8 | vander | 求解范德蒙线性方程 |
| 2.8 | toeplz | 求解托普雷兹线性方程 |
| 2.9 | choldc | 乔莱斯基分解 |
| 2.9 | cholsl | 乔莱斯基回代过程 |
| 2.10 | qrdcmp | QR 分解 |
| 2.10 | qrsolv | QR 回代过程 |
| 2.10 | rsolv | 右三角回代过程 |
| 2.10 | qrupdt | 最新的 QR 分解程序 |
| 2.10 | rotate | 用于 **qrupdt** 的雅可比旋转 |
| 3.1 | polint | 多项式插值 |
| 3.2 | ratint | 有理函数插值 |
| 3.3 | spline | 构造三次样条 |
| 3.3 | splint | 三次样条插值 |
| 3.4 | locate | 用二分法搜索有序表 |
| 3.4 | hunt | 当调用是相关时搜索有序表 |
| 3.5 | polcoe | 从列表值求多项式系数 |
| 3.5 | polcof | 从列表值求多项式系数 |
| 3.6 | polin2 | 二维多项式插值 |
| 3.6 | bcucof | 构造二维双三次式 |
| 3.6 | bcuint | 二维双三插值 |
| 3.6 | splie2 | 构造二维样条 |
| 3.6 | splin2 | 二维样条插值 |
| 4.2 | trapzd | 梯形法则 |
| 4.2 | qtrap | 用梯形法则求积分 |
| 4.2 | qsimp | 用辛普胜法则求积分 |
| 4.3 | qromb | 用龙贝格自适应法求积分 |
| 4.4 | midpnt | 扩展中点法则 |
| 4.4 | qromo | 用开型龙贝格自适应法求积分 |
| 4.4 | midinf | 在半无穷区间上对函数积分 |
| 4.4 | midsql | 对具有下限平方根奇异点的函数积分 |
| 4.4 | midsqu | 对具有上限平方根奇异点的函数积分 |
| 4.4 | midexp | 对呈指数衰减的函数积分 |
| 4.5 | qgaus | 高斯求积法对函数积分 |
| 4.5 | gauleg | Gauss-Legendre 的权和横坐标 |
| 4.5 | gaulag | Gauss-Laquerre 的权和横坐标 |
| 4.5 | gauher | Gauss Hermite 的权和横坐标 |
| 4.5 | gaujac | Gauss-Jacobi 的权和横坐标 |
| 4.5 | gaucof | 从正交多项式求积分的权 |
| 4.5 | orthog | 构造非经典正交多项式 |
| 4.6 | quad3d | 在三维空间上对函数求积分 |

| | | |
|---|---|---|
| 5.1 | eulsum | Euler-Van Wijngarrden 算法求级数和 |
| 5.3 | ddpoly | 计算多项式和它的导数 |
| 5.3 | poldiv | 用一多项式除另一多项式 |
| 5.3 | ratval | 有理函数求值 |
| 5.7 | dfridr | Ridders 方法的数值求导 |
| 5.8 | chebft | 用切比雪夫多项式拟合一个函数 |
| 5.8 | chebev | 切比雪夫多项式求值 |
| 5.9 | chder | 对已作切比雪夫拟合的函数求导 |
| 5.9 | chint | 对已作切比雪夫拟合的函数求积 |
| 5.10 | chebpc | 切比雪夫拟合的多项式系数 |
| 5.10 | pcshft | 位移多项式的多项式系数 |
| 5.11 | pccheb | 程序 **chebpc** 的逆过程 |
| 5.12 | pade | 从幂级数系数作 pade 逼近 |
| 5.13 | ratlsq | 最小二乘法的有理数拟合 |
| | | |
| 6.1 | gammln | Γ函数的对数 |
| 6.1 | factrl | 阶乘函数 |
| 6.1 | bico | 二项式系数函数 |
| 6.1 | factln | 阶乘函数的对数 |
| 6.1 | beta | B 函数 |
| 6.2 | gammp | 不完全Γ函数 |
| 6.2 | gammq | 互补不完全Γ函数 |
| 6.2 | gser | 用级数求不完全Γ函数 |
| 6.2 | gcf | 用连分式求不完全Γ函数 |
| 6.2 | erff | 误差函数 |
| 6.2 | erffc | 互补误差函数 |
| 6.2 | erfcc | 互补误差函数,简要的程序 |
| 6.3 | expint | 指数积分 $E_n$ |
| 6.3 | ei | 指数积分 $E_i$ |
| 6.4 | betai | 不完全 B 函数 |
| 6.4 | betacf | 用於 **betai** 的连分式求值 |
| 6.5 | bessj0 | 贝塞尔函数 $J_0$ |
| 6.5 | bessy0 | 贝塞尔函数 $Y_0$ |
| 6.5 | bessj1 | 贝塞尔函数 $J_1$ |
| 6.5 | bessy1 | 贝塞尔函数 $Y_1$ |
| 6.5 | bessy | 整数阶贝塞尔函数 $Y$ |
| 6.5 | bessj | 整数阶贝塞尔函数 $J$ |
| 6.6 | bessi0 | 修正的贝塞尔函数 $I_0$ |
| 6.6 | bessk0 | 修正的贝塞尔函数 $K_0$ |
| 6.6 | bessi1 | 修正的贝塞尔函数 $I_1$ |
| 6.6 | bessk1 | 修正的贝塞尔函数 $K_1$ |
| 6.6 | bessk | 修正的整数阶贝塞尔函数 $K$ |
| 6.6 | bessi | 修正的整数阶贝塞尔函数 $I$ |
| 6.7 | bessjy | 分数阶贝塞尔函数 |
| 6.7 | beschb | 用于 **bessjy** 的切比雪夫展开式 |
| 6.7 | bessik | 修正的分数阶贝塞尔函数 |
| 6.7 | airy | Airy 函数 |
| 6.7 | sphbes | 球面贝塞尔函数 $j_n$ 和 $y_n$ |
| 6.8 | plgndr | 连带的勒让德多项式(球面调和函数) |
| 6.9 | frenel | Fresnel 积分 $S(x)$ 和 $C(x)$ |
| 6.9 | cisi | 余弦和正弦积分 $Ci$ 和 $Si$ |
| 6.10 | dawson | Dawson 积分 |
| 6.11 | rf | 第一类卡尔森椭圆积分 |
| 6.11 | rd | 第二类卡尔森椭圆积分 |
| 6.11 | rj | 第三类卡尔森椭圆积分 |
| 6.11 | rc | 卡尔森退化椭圆积分 |
| 6.11 | ellf | 第一类勒让德椭圆积分 |
| 6.11 | elle | 第二类勒让德椭圆积分 |
| 6.11 | ellpi | 第三类勒让德椭圆积分 |
| 6.11 | sncndn | 雅可比椭圆函数 |
| 6.12 | hypgeo | 复超几何函数 |
| 6.12 | hypser | 复超几何函数,级数求值 |
| 6.12 | hypdrv | 复超几何函数的导数 |
| | | |
| 7.1 | ran0 | 用 Park 和 Miller 最低标准生成随机偏离 |
| 7.1 | ran1 | 随机偏离,最低标准加混洗 |
| 7.1 | ran2 | 随机偏离,L'Ecuyer 长周期的混洗 |
| 7.1 | ran3 | 随机偏离,Knuth 相减法 |
| 7.2 | expdev | 指数随机偏离 |
| 7.2 | gasdev | 正态分布的随机偏离 |
| 7.3 | gamdev | Γ分布的随机偏离 |
| 7.3 | poidev | 泊松分布的随机偏离 |
| 7.3 | bnldev | 二项分布的随机偏离 |
| 7.4 | irbit1 | 随机位序列 |
| 7.4 | irbit2 | 随机位序列 |
| 7.5 | psdes | 64 位混洗的伪数据加密标准 |
| 7.5 | ran4 | 使用数据加密标准混洗生成的随机偏离 |
| 7.7 | sobseq | Sobol 准随机序列 |
| 7.8 | vegas | 自适应多维蒙特卡罗积分 |
| 7.8 | rebin | 用于 **vegas** 程序中取样重新分级 |
| 7.8 | miser | 递归多维蒙特卡罗积分 |

| | | |
|---|---|---|
| 7.8 | ranpt | 获取随机点，用于**miser**程序 |
| 8.1 | piksrt | 用直接插入法对数组排序 |
| 8.1 | piksr2 | 用直接插入法对两个数组排序 |
| 8.1 | shell | 用剥壳法对数组排序 |
| 8.2 | sort | 用快速排序法对数组排序 |
| 8.2 | sort2 | 用快速排序法对两个数组排序 |
| 8.3 | hpsort | 用堆积排序法对数组排序 |
| 8.4 | indexx | 构造数组的索引表 |
| 8.4 | sort3 | 排序，使用索引表对3个或多个数组排序 |
| 8.4 | rank | 构造数组的秩表 |
| 8.5 | select | 寻找数组中第 $N$ 大的元素 |
| 8.5 | selip | 找出数组中第 $N$ 大的元素，而不变动原数组 |
| 8.5 | hpsel | 找出 $M$ 个大值，而不变动原数组 |
| 8.6 | eclass | 从表确定等价类 |
| 8.6 | eclazz | 由过程确定等价类 |
| 9.0 | scrsho | 画函数图形以搜索根 |
| 9.1 | zbrac | 为求根向外搜索划界 |
| 9.1 | zbrak | 为求根向内搜索划界 |
| 9.1 | rtbis | 用二分法寻找函数的根 |
| 9.2 | rtflsp | 用试位法寻找函数的根 |
| 9.2 | rtsec | 用弦截法寻找函数的根 |
| 9.2 | zriddr | 用里德的方法寻找函数的根 |
| 9.3 | zbrent | 用布伦特的方法寻找函数的根 |
| 9.4 | rtnewt | 用牛顿-拉斐森算法寻找函数的根 |
| 9.4 | rtsafe | 用牛顿-拉斐森和二分法寻找函数的根 |
| 9.5 | laguer | 用拉盖尔的方法找多项式的根 |
| 9.5 | zroots | 用具有降阶的拉盖尔方法求多项式的根 |
| 9.5 | zrhqr | 特征值法求多项式的根 |
| 9.5 | qroot | 多项式的复根或重根，贝尔斯托法 |
| 9.6 | mnewt | 牛顿法解方程组 |
| 9.7 | lnsrch | 用于**newt**的沿直线搜索 |
| 9.7 | newt | 多维牛顿法的全局收敛 |
| 9.7 | fdjac | 用于**newt**的有限差分雅可比行列式 |
| 9.7 | fmin | 用于**newt**的向量函数的范数 |
| 9.7 | broydn | 对于方程系统的弦截法 |
| 10.1 | mnbrak | 划界函数最小值 |
| 10.1 | golden | 用黄金分割搜索法寻找函数极小值 |
| 10.2 | brent | 用布伦特的方法寻找函数极小值 |
| 10.3 | dbrent | 利用导数信息寻找函数极小值 |
| 10.4 | amoeba | 用下降单纯形法求多维极小化 |
| 10.4 | amotry | 求轨迹点，用于**amoeba**程序中 |
| 10.5 | powell | 用鲍威尔的方法求多维极小化 |
| 10.5 | linmin | 沿多维空间一条射线求函数极小值 |
| 10.5 | f1dim | 用于**linmin**中的函数 |
| 10.6 | frprmn | 用共轭梯度法求多维极小化 |
| 10.6 | dlinmin | 利用导数信息沿一条射线求函数极小值 |
| 10.6 | df1dim | 用于**dlinmin**中的函数 |
| 10.7 | dfpmin | 用变度量法求多维极小化 |
| 10.8 | simplx | 线性函数的线性规划极大化 |
| 10.8 | simp1 | 用于**simplx**中的线性规划 |
| 10.8 | simp2 | 用于**simplx**中的线性规划 |
| 10.8 | simp3 | 用于**simplx**中的线性规划 |
| 10.9 | anneal | 用模拟退火法求解旅行销售员问题 |
| 10.9 | revcst | 反转的代价，用于**anneal** |
| 10.9 | reverse | 执行反转过程，用于**anneal** |
| 10.9 | trncst | 传送的代价，用于**anneal** |
| 10.9 | trnspt | 执行传送操作，用于**anneal** |
| 10.9 | metrop | 米托波利斯算法，用于**anneal** |
| 10.9 | amebsa | 连续空间中的模拟退火法 |
| 10.9 | amotsa | 计算一个试验点，用于**amebsa** |
| 11.1 | jacobi | 对称矩阵的特征值和特征向量 |
| 11.1 | eigsrt | 特征向量，按特征值顺序而排序 |
| 11.2 | tred2 | 实对称矩阵的豪斯贺德约化 |
| 11.3 | tqli | 对称三对角矩阵的特征解 |
| 11.5 | balanc | 配平一个非对称矩阵 |
| 11.5 | elmhes | 将一般矩阵约化成海森伯格型式 |
| 11.6 | hqr | 海森伯格矩阵的特征值 |
| 12.2 | four1 | 一维快速傅里叶变换(FFT) |
| 12.3 | twofft | 两个实函数的快速傅里叶变换 |
| 12.3 | realft | 单个实函数的快速傅里叶变换 |

| 12.3 | sinft | 快速正弦变换 |
|---|---|---|
| 12.3 | cosft1 | 带端点的快速余弦变换 |
| 12.3 | cosft2 | "交错"快速余弦变换 |
| 12.4 | fourn | 多维快速傅里叶变换 |
| 12.5 | rlft3 | 二维或三维实数据的FFT |
| 12.6 | fourfs | 存储在外部媒体的大数据集的FFT |
| 12.6 | fourew | 用于fourfs的重绕和置换文件 |
| 13.1 | convlv | 使用FFT作数据的卷积和解卷积 |
| 13.2 | correl | 使用FFT作数据的相关和自相关 |
| 13.4 | spctrm | 使用FFT作功率谱估计 |
| 13.6 | memcof | 对最大熵(MEM)系数求值 |
| 13.6 | fixrts | 多项式式的根,反射到单位圆内 |
| 13.6 | predic | 使用MEM系数的线性预测 |
| 13.7 | evlmem | 由MEM系数作功率谱估计 |
| 13.8 | period | 非均匀取样数据的功率谱 |
| 13.8 | fasper | 非均匀取样大数据集的功率谱 |
| 13.8 | spread | 利用**fasper**,对数组进行外插值 |
| 13.9 | dftcor | 为傅里叶积分计算端点系数 |
| 13.9 | dftint | 高精度的傅里叶积分 |
| 13.10 | wt1 | 一维离散小波变换 |
| 13.10 | daub4 | DAUb4系数的小波滤波器 |
| 13.10 | pwtset | 为程序**pwt**初始化系数 |
| 13.10 | pwt | 部分小波变换 |
| 13.10 | wtn | 多维离散小波变换 |
| 14.1 | moment | 计算数据集的矩 |
| 14.2 | ttest | 均值差的学生t检验 |
| 14.2 | avevar | 计算数据集的均值和方差 |
| 14.2 | tutest | 方差不相等情况,均值的学生t检验 |
| 14.2 | tptest | 配对数据情况,均值的学生$t$检验 |
| 14.2 | ftest | 对方差之差的$F$检验 |
| 14.3 | chsone | 数据和模型之间差异的$\chi^2$检验 |
| 14.3 | chstwo | 两数据集之间差异的$\chi^2$检验 |
| 14.3 | ksone | 数据相对模型的K-S检验 |
| 14.3 | kstwo | 两数据集之间的K-S检验 |
| 14.3 | probks | K-S概率函数 |
| 14.4 | cntab1 | 使用$\chi^2$的列联表分析 |
| 14.4 | cntab2 | 使用熵度量的列联表分析 |
| 14.5 | pearsn | 两数据集之间的皮尔逊相关 |
| 14.6 | spear | 两数据集之间的斯皮尔曼秩相关 |
| 14.6 | crank | 用数组元素的秩替换这些元素 |
| 14.6 | kendl1 | 两数据集之间的相关,肯德尔τ |
| 14.6 | kendl2 | 使用肯德尔τ的列联表分析 |
| 14.7 | ks2d1s | 数据对模型的二维K-S检验 |
| 14.7 | quadct | 计算各象限中的点,用于**ks2d1s** |
| 14.7 | quadvl | 象限的概率,用于**ks2d1s** |
| 14.7 | ks2d2s | 数据对数据的二维K-S检验 |
| 14.8 | savgol | S-G平滑系数 |
| 15.2 | fit | 最小二乘法将数据拟合成直线 |
| 15.3 | fitexy | 拟合数据成直线,x和y两者都有误差 |
| 15.3 | chixy | 计算$\chi^2$,用于**fitexy** |
| 15.4 | lfit | 用正规方程的一般线性最小二乘拟合 |
| 15.4 | covsrt | 重排协方差矩阵,用于**lfit** |
| 15.4 | svdfit | 用奇异值分解法的一般线性最小二乘拟合 |
| 15.4 | svdvar | 来自奇异值分解的方差 |
| 15.4 | fpoly | 用**lfit**或**svdfit**拟合多项式 |
| 15.4 | fleg | 用**lfit**或**svdfit**拟合勒让德多项式 |
| 15.5 | mrqmin | 马阔特方法,非线性最小二乘拟合 |
| 15.5 | mrqcof | 用于**mrqmin**中计算系数 |
| 15.5 | fgauss | 使用**mrqmin**拟合高斯分布的和 |
| 15.7 | medfit | 稳健地将数据拟合成直线,最小绝对偏差 |
| 15.7 | rofunc | 用于**medfit**中稳健地拟合数据 |
| 16.1 | rk4 | 用四阶龙格-库塔法计算ODEs的一步积分 |
| 16.1 | rkdumb | 用四阶龙格-库塔法对ODEs积分求解 |
| 16.2 | rkqs | 具有精确度监控的ODEs的一步积分 |
| 16.2 | rkck | 用于**rkqs**的凯希-卡普-龙格-库塔步 |
| 16.2 | odeint | 具有精确度监控的ODEs积分 |
| 16.3 | mmid | 用修正中点法的ODEs积分 |

| | | |
|---|---|---|
| 16.4 | bsstep | 用布里斯奇-斯托步积分的积分 |
| 16.4 | pzextr | 用于 **bsstep** 的多项式外推 |
| 16.4 | rzextr | 用于 **bsstep** 的有理函数外推 |
| 16.5 | stoerm | 二阶守恒 ODEs 的积分 |
| 16.6 | stiff | 用四阶罗塞布鲁克积分刚性的 ODEs |
| 16.6 | jacobn | 用于 **stiff** 的雅可夫程序的范例 |
| 16.6 | derivs | 用于 **stiff** 的求导数程序的范例 |
| 16.6 | simpr | 用半隐式中点法积分刚性的 ODEs |
| 16.6 | stifbs | 用布里斯奇-斯托步积分刚性 ODEs |
| 17.1 | shoot | 打靶法求解两点边界值问题 |
| 17.2 | shootf | 对合适点打靶求解两点边界值问题 |
| 17.3 | solvde | 用松驰法求解两点边界值问题 |
| 17.3 | bksub | 回代,用于 **solvde** |
| 17.3 | pinvs | 对角化子块,用于 **solvde** |
| 17.3 | red | 约化矩阵的列,用于 **slovde** |
| 17.4 | sfroid | 用 **solvde** 计算球体调和函数 |
| 17.4 | difeq | 球体矩阵系数,用于 **sfroid** |
| 17.4 | sphoot | 用 **shoot** 计算球体调和函数 |
| 17.4 | sphfpt | 用 **shootf** 计算球体调和函数 |
| 18.1 | fred2 | 求解第二类线性弗雷德霍姆方程 |
| 18.1 | fredin | 和 **fred2** 一起得到内插解 |
| 18.2 | voltra | 第二类线性沃尔泰拉方程 |
| 18.3 | wwghts | 任意奇异核的积分权重 |
| 18.3 | kermom | 对于一个奇异核的矩的样本程序 |
| 18.3 | quadmx | 积分矩阵的样本程序 |
| 18.3 | fredex | 求解奇异弗雷德霍姆方程的范例 |
| 19.5 | sor | 用逐次超松驰法求解椭圆 PDE |
| 19.6 | mglin | 用多网格法求解线性椭圆 PDE |
| 19.6 | rstrct | 半加权限制,用于 **mglin**,**mgfas** |
| 19.6 | interp | 双线性拓展,用于 **mglin**,**mgfas** |
| | | 且,用于 **mglin** |
| | | 上求解,用于 **mglin** |
| | | 得尔松驰,用于 **mglin** |
| 19.6 | resid | 计算残差,用于 **mglin** |
| 19.6 | copy | 用于 **mglin** 和 **mgfas** 的实用程序 |
| 19.6 | fill0 | 用于 **mglin** 的实用程序 |
| 19.6 | mgfas | 用多网格法求解非线性椭圆 PDE |
| 19.6 | relax2 | 高斯-赛得尔松驰,用于 **mgfas** |
| 19.6 | slvsm2 | 最粗网格上求解,用于 **mgfas** |
| 19.6 | lop | 应用非线性操作,用于 **mgfas** |
| 19.6 | matadd | 用于 **mgfas** 的实用程序 |
| 19.6 | matsub | 用于 **mgfas** 的实用程序 |
| 19.6 | anorm2 | 用于 **mgfas** 的实用程序 |
| 20.1 | machar | 诊断计算机的浮点运算 |
| 20.2 | igray | 格雷码和它的逆 |
| 20.3 | icrc1 | 用于 **icrc** 的循环冗余度校验程式 |
| 20.3 | icrc | 循环冗余度校验程式 |
| 20.3 | decchk | 十进制校验位的计算或证实 |
| 20.4 | hufmak | 构造一个霍夫曼码 |
| 20.4 | hufapp | 追加比特位到霍夫曼码,用于 **hufmak** |
| 20.4 | hufenc | 用霍夫曼码编码和压缩一个字符 |
| 20.4 | hufdec | 用霍夫曼码译码和解压缩一个字符 |
| 20.5 | arcmak | 构造一个算术码 |
| 20.5 | arcode | 用算术码编码或译码一个字符 |
| 20.5 | arcsum | 将整数加入到字节串中,用于 **ar code** |
| 20.6 | mpops | 多精度算术运算,简单的运算 |
| 20.6 | mpmul | 多精度乘法,采用 FFT 方法 |
| 20.6 | mpinv | 多精度的倒数 |
| 20.6 | mpdiv | 多精度的除法和余数 |
| 20.6 | mpsqrt | 多精度的平方根 |
| 20.6 | mp2dfr | 多精度转换成十进制数 |
| 20.6 | mppi | 多精度的示例,计算 π 的多位数字 |